Comprehensive and Practical Policy Compliance in Data Retrieval Systems

A dissertation submitted towards the degree

Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Eslam Elnikety

Saarbrücken, 2019

Day of Colloquium	24 June 2020
Dean of the Faculty	Univ.-Prof. Dr. Thomas Schuster
Chair of the Committee	Prof. Dr. Andreas Zeller
Reporters	
First Reviewer	Prof. Dr. Peter Druschel
Second Reviewer	Dr. Deepak Garg
Third Reviewer	Prof. Dr. Paul Francis
Fourth Reviewer	Dr. Saikat Guha
Academic Assistant	Dr. Isaac Sheff

Abstract

Data retrieval systems such as online search engines and online social networks process many data items coming from different sources, each subject to its own data use policy. Ensuring compliance with these policies in a large and fast-evolving system presents a significant technical challenge since bugs, misconfigurations, or operator errors can cause (accidental) policy violations. To prevent such violations, researchers and practitioners develop policy compliance systems.

Existing policy compliance systems, however, are either not comprehensive or not practical. To be comprehensive, a compliance system must be able to enforce users' policies regarding their personal privacy preferences, the service provider's own policies regarding data use such as auditing and personalization, and regulatory policies such as data retention and censorship. To be practical, a compliance system needs to meet stringent requirements: *(1)* runtime overhead must be low; *(2)* existing applications must run with few modifications; and *(3)* bugs, misconfigurations, or actions by unprivileged operators must not cause policy violations.

In this thesis, we present the design and implementation of two comprehensive and practical compliance systems: **T**HOTH and **S**HAI. **T**HOTH relies on pure runtime monitoring: it tracks data flows by intercepting processes' I/O, and then it checks the associated policies to allow only policy-compliant flows at runtime. **S**HAI, on the other hand, combines offline analysis and light-weight runtime monitoring: it pushes as many policy checks as possible to an offline (flow) analysis by predicting the policies that data-handling processes will be subject to at runtime, and then it compiles those policies into a set of fine-grained I/O capabilities that can be enforced directly by the underlying operating system.

Kurzdarstellung

Datenanalysesysteme verarbeiten Dateneinträge aus verschieden Quellen, die jeweils Datennutzungsregeln unterliegen. Regelkonformietät in einem großen und schnell-weiterentwickeln dem System zu garantieren, stellt große technische Herausforderung dar, insbesondere da Softwarefehler, falsche Konfigurationen oder Bedienfehler (versehentlichen) Regelbruch verursachen können. Um diesen Regelbruch zu verhindern, entwickeln Forscher und Entwickler regelkonforme Systeme.

Existierende regelkonforme Systeme sind jedoch nicht allumfassend und praktikabel. Ein allumfassendes System muss in der Lage sein, benutzerspezifische, dienstleisterspezifische oder gesetzliche Regeln zu berücksichtigen. Diese berücksichtigen die Privatsphäreeinstellungen eines Nutzers, ermöglichen die Überprüfung oder Personalisierung durch den Dienstleister oder garantieren den Datenerhalt oder Zensur. Ein praktikables System hingegen hat strikte Anforderungen: (1) Die Laufzeitkosten sind gering; (2) existierende Anwendungen erfordern wenige Anpassungen; (3) Softwarefehler, falsche Konfigurationen oder Bedienfehler werden erkannt und der Regelbruch wird verhindert.

Diese Doktorarbeit stellt das Konzept und die Implementierung von zwei allumfassend, praktikable, regelkonforme Systemen vor: **T**HOTH und **S**HAI. **T**HOTH stützt sich auf die Überwachung zur Laufzeit. Kontinuierlich verfolgt es die Eingabedaten und den Datenfluss zwischen Prozessen und evaluiert assoziierten Regeln, um nur regelkonforme Datenflüsse zuzulassen. Demgegenüber kombiniert **S**HAI eine offline Flussanalyse mit einer vereinfachten Laufzeitüberwachung. Dafür werden die meisten Evaluierungen bereits vor der Ausführung durch die offline Flussanalyse evaluiert, indem alle Prozesse und deren Eingabedaten analysiert werden. Diese Analyse erzeugt detaillierte Zugriffsberechtigungen, die vom Betriebssystem berücksichtigt werden.

Publications

Parts of this thesis have appeared in the following publications:

- "Shai: Enforcing Data-Specific Policies with Near-Zero Runtime Overhead". Eslam Elnikety, Deepak Garg, and Peter Druschel. In *CoRR* abs/1801.04565 (2018). Available at `https://arxiv.org/abs/1801.04565`.
- "Thoth: Comprehensive Policy Compliance in Data Retrieval Systems". Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. In the proceedings of the 25th USENIX Security Symposium (USENIX Security'16), Austin, TX, Aug. 2016.

Additional publications while at MPI-SWS:

- "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)". Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Deepak Garg, and Peter Druschel. In the proceedings of the 28th USENIX Security Symposium (USENIX Security'19), Santa Clara, CA, Aug. 2019
- "Qapla: Policy compliance for database-backed systems". Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. In the proceedings of the 26th USENIX Security Symposium (USENIX Security'17), Vancouver, BC, Canada, Aug. 2017.
- "Light-weight Contexts: An OS Abstraction for Safety and Performance". James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. In the proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16), Savannah, GA, Nov. 2016.
- "Guardat: Enforcing data policies at the storage layer". Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Ansley Post, Rodrigo Rodrigues, and Johannes Gehrke. In the proceedings of the 10th European Conference on Computer Systems (EuroSys'15), Bordeaux, France, Apr. 2015.
- "Protecting Data Integrity with Storage Leases". Anjo Vahldiek, Eslam Elnikety, Ansley Post, Peter Druschel, and Rodrigo Rodrigues. Technical Report 2011-008, MPI-SWS, 2011.

Acknowledgements

The research work of this thesis would not have been possible if it were not for the help of numerous others during my time as a graduate student at MPI-SWS. First and foremost, I thank Peter Druschel and Deepak Garg. They have been quite the advising team. I am greatly indebted to them for their mentoring, constructive criticism, and many hours of stimulating and enlightening discussions.

I have been fortunate to work closely with Aastha Mehta and Anjo Vahldiek-Oberwagner. They have been fantastic colleagues. Besides creating a fun working environment, they helped a lot with the software produced in this thesis. I am thankful for my co-authors, in particular Bobby Bhattacharjee and James Litton, for their contagious energy. My colleagues in the Sys-Nets group, in particular Paarijaat Aditya and Pedro Fonseca, produced an inspiring research environment and often provided invaluable feedback to this research.

Finally, I thank my family and friends for their unwavering support and love.

Contents

1 Introduction

This chapter introduces data retrieval systems and some of the data use policies that govern how these systems ought to collect, process, and serve data. To enforce such policies, researchers and practitioners develop policy compliance systems, which we cover briefly in this chapter. The discussion regarding the existing policy compliance systems in this chapter is at a high level and is not meant to be exhaustive. We provide only an overview to highlight the distinguishing features of the compliance systems presented in this thesis, and we defer further details to later chapters.

1.1 Data Retrieval Systems and Data Use Policies

Data retrieval systems are a class of online services that store, aggregate, index, recommend, and serve information. This class includes large-scale social media sites, online search engines, e-commerce sites, and numerous organizational, corporate, and government information services. Examples include large providers like Amazon, Facebook, eBay, Google, Microsoft, and numerous smaller, domain-specific sharing, trading and networking sites.

Data retrieval systems typically serve a searchable corpus of documents, web pages, blogs, personal e-mails, online social network profiles and posts, along with real-time microblogs, advertisements, stocks and news tickers. Figure 1.1 provides a figurative depiction of data retrieval systems, showing an example of a typical search engine and its different data sources and processing pipelines. The example search engine stores personal messages, e-mails, and social network profiles and posts. It also collects public information such as web pages and real-time data streams. The search engine makes data from these different sources available to users (external clients and the provider's own employees) via a search pipeline, where users submit queries and receive mash-ups of data items from different sources. The search engine also collects users' queries and click streams (e.g., documents viewed) to personalize search results. Additionally, the search engine uses an advertisement service to generate revenue.

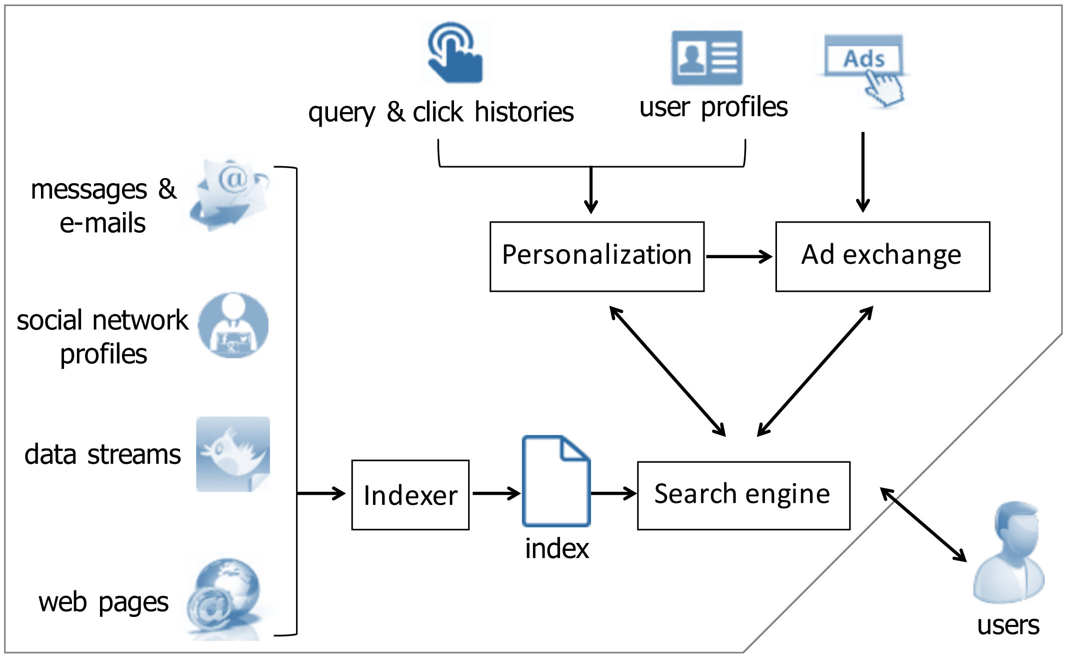


FIGURE 1.1: An example search engine with processing pipelines for searching, personalization, and advertisements.

As the example shows, data retrieval systems aggregate, index, recommend, and serve many data items from different sources. Each data item served or used by a data retrieval system may have its own usage policy. For instance, e-mail is private to its sender/receiver(s), online social network data and blogs may be restricted to friends, and corporate documents are limited to (authorized) employees. Those are a few examples of the access control settings that data items may be subject to. Additionally, systems need to comply with local laws and regulations, which may require them, for instance, to filter certain data items within a given jurisdiction to enforce censorship. Moreover, the system must comply with the provider's own privacy policy, which may stipulate, for instance, that a user's query and click streams be used only for personalization. Finally, other usage policies may restrict the use of specific (meta)data, require expiration, or permit access subject to logging.

Complying with data usage policies is crucial. Policy violations and data breaches constitute a serious threat [92], and many incidents get media coverage [72, 71, 84, 83, 93]. (Privacy Rights Clearinghouse reports over 8,000 data breaches made public since 2005 [70].) Noncompliance may lead to loss of reputation and customer confidence. The stakes are high: providers may face stiff fines and legal repercussions in case of policy violations. For example, under the EU General Data Protection Regulation (GDPR), organizations in breach may be fined up to 4% of the previous financial year's annual revenue or €20 Million, whichever is greater [82]. Similarly, US companies are legally required to publish their data use policies and are subject to penalties in cases of violation [78]. Hence, providers have a vested interest in ensuring compliance with all applicable data use policies.

1.2 Policy Compliance Systems: An Overview

Ensuring compliance with all applicable data use policies in data retrieval systems presents a significant technical challenge. First, there are many data items and (possibly as many) data use policies. Moreover, the policy in effect for a data item may depend on checks, settings, and configurations in the many components and the several layers of the software stack, including file and operating systems, runtime frameworks, and application code. These scattered and numerous checks, settings, and configurations make it difficult to audit and reason about policy compliance. Second, policy violations can be caused by misconfigurations, bugs, and vulnerabilities in the data retrieval systems' software stacks, which are large and complex.

Traditionally, policy compliance in data retrieval systems is best-effort. Providers rely on manual reviews and audits, which lack coverage and fall short of providing (any) policy compliance guarantees [75]. Moreover, such manual reviews and audits are time-consuming and labor-intensive. Therefore, many small- and medium-scale providers cannot afford such practices given their limited IT budgets. While that might not be the case for large-scale providers who employ hundreds or thousands of software engineers and privacy officers, those providers have additional challenges. Large providers typically have fast-evolving application codebases, with frequent updates to improve performance or to introduce new features. When policy compliance checks are entangled with dynamic and agile application code, the policies in effect are difficult to maintain. Moreover, misconfigurations or application updates that introduce bugs can cause policy violations.

Hence, developing technical mechanisms for *ensuring* policy compliance in data retrieval systems is important. In fact, there has been significant work on ensuring policy compliance by both practitioners and researchers. For instance, Microsoft's Bing relies on the Grok system to prevent policy violations in the search engine's back-end [75], Facebook relies on the IVD system to prevent missing or incorrect authorization checks from violating policies [56], besides many research projects that can enforce data policies such Flume [49], HiStar [101], and Asbestos [31] .

Existing policy compliance systems and mechanisms, however, either lack comprehensiveness or are impractical for data retrieval systems. We explain the lack of comprehensiveness and the impracticality next.

Comprehensiveness. For a compliance system to be comprehensive, it must be able to express and enforce *individual* policies. An individual policy is specific to a given data item or to a given user's data items. In fact, many of the data use policies that arise in practice are individual policies. For instance, Alice's e-mails are private to Alice, whereas Bob's e-mails are private to Bob. A particular blog post of Alice's may be public, whereas another of hers may be available only to her friends. These are example individual policies that capture personal access control settings which are user- and item-specific. Other examples of individual policies capture legal and contractual regulations. For instance, a given data item may not be accessed in a given jurisdiction to comply with the censorship dictated by the local law of the respective jurisdiction. A data item in a data stream (e.g., news ticker) may expire within 48 hours, whereas it may be permissible to retain another item in the same stream indefinitely. Finally, other example individual policies reflect the provider's own privacy policy, which may stipulate, for instance, that the query and click streams of a given user be used only for personalization subject to the user opting-in.

All the previously mentioned policies are examples of individual policies, constituting an important class of data use policies. In fact, compliance with some individual policies is required by law. For instance, the EU General Data Protection Regulation (GDPR) explicitly grants users individual choice regarding the use of their personal data [82].

Enforcing individual policies requires dynamic analysis since individual policies are not amenable to pure static analysis techniques for two primary reasons. First, static analysis loses precision quickly under such policies. The same front-end program that Alice's friend, say Carol, uses to access the ("visible only to Alice's friends") blog post is used by other clients who are not necessarily Alice's friends. Thus, a given program variable may contain data with very different individual policies over time at the same program point and, hence, the abstraction of static analysis may lose precision quickly. Second, individual policies often refer to information available only at runtime and cannot be resolved statically. For example, whether an access is on behalf of Carol is available only during runtime from the session information. Besides user's identity, other runtime information which data use policies often refer to include the geographic location of a connected client (e.g., when a data item is censored in a specific jurisdiction), wall-clock time (e.g., when a news ticker item expires at a specific time), or content state (e.g., when a log entry must exist before accessing sensitive content).

No existing work covers the important class of individual policies. Existing policy compliance systems for data retrieval systems cannot enforce individual policies since they rely on static analysis techniques, while enforcing individual policies requires dynamic analysis. Such compliance systems usually target *column-specific policies*. In contrast to individual policies which are per-user and data-specific, column-specific policies apply uniformly to all data of a specific type, e.g., the policy "no IP address can be used for advertizing." For such policies, pure static analysis techniques may suffice. For instance, the Grok system combines light-weight static analysis with heuristics to annotate source code in order to check for violations against column-specific policies in Bing's back-end [75]. However, Grok cannot check violations against individual policies for the reasons described earlier.

The missing comprehensive enforcement with regard to individual policies is a significant and important part of policy enforcement that the compliance systems developed in this thesis address.

Practicality. For a compliance system to be practical, it needs to meet the following requirements.

(1) Low runtime overhead. A policy compliance system must have low runtime overhead to be practical. High runtime overheads on the critical path (e.g., when serving users' requests) may prohibit adoption, especially in medium- to large-scale data retrieval systems. Runtime overheads directly increase the providers' operational cost, as providers either need to sacrifice valuable runtime cycles for policy enforcement or need to provision more resources to sustain performance (i.e., a specific throughput rate).

(2) Compatibility with existing systems. To be practical, a compliance system must be compatible with existing applications and software systems, requiring few modifications to the existing software stack and application codebases. Otherwise, providers will have to invest heavy resources and engineering hours to adapt (and maybe re-write) the complex data retrieval processing pipelines to the software stack required by the compliance system — which tends to be impractical.

(3) Compliance despite application bugs, misconfigurations, and errors by unprivileged operators. Given the complexity of the processing pipelines within data retrieval systems, such systems are vulnerable to bugs, misconfigurations, and operator errors. To be practical, the compliance system must enforce policy despite these vulnerabilities. Therefore, policy specification and enforcement should be entirely independent of the application codebase.

As we have mentioned earlier, enforcing individual policies requires dynamic analysis since individual policies are not amenable to pure static analysis techniques. There are existing dynamic analysis systems which can enforce individual policies (and can, in principle, satisfy the comprehensiveness requirement). These systems fall into two broad classes: dynamic analysis for *access control* and for *flow control*. The former is categorically not suitable for policy compliance in data retrieval systems, whereas the existing systems that fall into the latter class fail to meet one (or more) of the practicality requirements needed for policy compliance in data retrieval systems.

Access control systems ensure that access conditions are satisfied before releasing data. Examples of dynamic analysis systems for access control include Guardat [87], Taos [94], and PCFS [39]. Generally, access control systems do not track data flows and, therefore, offer only one-point enforcement. Such systems are not suitable for data retrieval systems which have processing pipelines with (multiple) intermediate steps.

Information flow control (IFC) systems restrict a program's data flow to enforce (flow) policies. Such systems are suitable for the pipelined nature of data retrieval systems. Nonetheless, the existing systems that fall into this category fail to meet one (or more) of the practicality requirements needed for policy compliance in data retrieval systems. For instance, the OS-level dynamic flow analysis techniques in Flume [49], HiStar [101], and Asbestos [31] can enforce individual policies. However, since these systems use abstract labels as taint, they rely on trusted application components to map between labels and access policies and to declassify data. Hence, the correctness of policy configuration and enforcement relies on those trusted application components —violating practicality requirement *(3)*. Moreover, HiStar and Asbestos are non-standard operating systems —violating practicality requirement *(2)*. Other systems rely on fine-grained dynamic flow analysis within the language runtime, such as RESIN [98] and COWL [81]. Fine-grained dynamic flow analysis incurs high runtime overhead and is language-dependent —violating practicality requirements *(1)* and *(2)*. Note that ensuring policy compliance in *data retrieval systems* is not a primary goal of the aforementioned systems, which otherwise offer practical solutions in their respective domains.

To summarize, existing policy compliance systems for data retrieval systems are either not comprehensive or are impractical. This is the gap that the work developed in this thesis fills.

1.3 Comprehensive and Practical Policy Compliance Systems

The goal of this thesis is to build comprehensive and practical policy compliance techniques for data retrieval systems. In this section, we first present, at a high level, key principles that guide the design of the systems developed in this thesis. Then, we describe two compliance systems developed in this work in further detail.

Item-specific policies. As explained earlier, many of the policies that arise in practice are user- and item-specific. Therefore, we would like to allow any data conduit —a file, key-value tuple, named pipe or network connection— to have its own data use policy.

Concise, declarative policies. We would like a policy attached to a conduit to be a *complete, and one point description* of all the confidentiality (i.e., conditions to read or disseminate) and integrity (i.e., conditions to update) requirements in effect for the data in that conduit. A policy is specified using a declarative language, separate from application code. This is in contrast to policies being implicitly specified in code and configuration of multiple software layers and large application codebases.

Enforcement at task boundary. Data retrieval systems tend to be implemented as a set of pipelined (and parallel) tasks. Enforcing data use policies at the tasks' boundaries is a natural match for this architecture, can be made efficient (as we show in this thesis), and requires few changes to existing services.

OS-level enforcement. Applications within data retrieval systems are complex and are constantly subject to ongoing development by an army of software engineers, which inevitably leads to bugs and errors. By comparison, the operating system is smaller, evolves slowly, and is maintained by a small team of experts. Therefore, the operating system is a suitable choice for policy enforcement in data retrieval systems.

1.3.1 THOTH

THOTH [33] is a kernel-level policy compliance layer. It performs coarse-grained runtime flow tracking to enforce data use policies. We give a high-level description of THOTH in the following.

In THOTH, the *provider* attaches policies directly to data items. A policy attached to a data item specifies all the confidentiality and integrity requirements in effect for that data item.

THOTH maps the tasks of a data retrieval system to OS processes, and implements a reference monitor (RM) in the kernel that intercepts tasks I/O and disallows data flows that would otherwise violate data confidentiality or integrity. To ensure confidentiality, the RM maintains a *taint* per process, which is the set of policies of all the data items a process has consumed in the past. The RM allows a process to produce output under two scenarios: *(1)* if the output is associated with a policy that is at least as restrictive as the process's taint, representing standard policy propagation; or *(2)* if the process's taint permits such output, allowing data declassification when extricating data from the system or when policy changes along the data flow path. These checks ensure that data flows never violate data confidentiality, since either the confidentiality restrictions on the data flow sources are carried forward, or the necessary conditions to declassify data as dictated by the policies are satisfied. Ensuring integrity is straightforward. A process is allowed to write/update a data item if the write does not violate the integrity requirement as stated in the item's policy.

We have implemented a THOTH prototype, consisting of a Linux kernel module that plugs in the Linux Security Module (LSM) [95] interface and couples with a userspace RM process that implements the taint tracking and policy evaluation logic. THOTH's runtime coarse-grained flow control incurs an overhead, but we show that the overhead is not too high for data retrieval systems that need to sustain a few hundred requests/second/machine. We measure an overhead of 3.6% on query throughput (with sufficiently long user sessions) in a search pipeline based on the widely used search engine Apache Lucene [3]. While this overhead may be too high for large-scale data retrieval systems that need to sustain higher request rate, we believe that it is suitable for many domain-specific data retrieval systems run by organizations, enterprises and governments.

To summarize, THOTH is a kernel-level policy compliance layer suitable for small- to medium-throughput data retrieval systems. It tracks and controls data flows across processes by intercepting I/O in the kernel. It prevents data leaks and corruption due to bugs and misconfigurations in application components, as well as actions by unprivileged operators.

1.3.2 SHAI

SHAI [32] is a direct improvement on THOTH, with the goal of reducing the runtime overhead of ensuring policy compliance in order to offer a compliance layer that is suitable for large-scale data retrieval systems. To motivate SHAI's design, we first outline at a high level the main sources of THOTH's runtime overhead.

THOTH's runtime overhead sources. As we mentioned earlier, THOTH controls data flows at runtime by *intercepting I/O* in the kernel and maintaining *per-process taint*.

- I/O interception is expensive. An interception involves a context switch to the userspace RM process. Also, the RM needs to maintain taint and to evaluate policies, consuming valuable runtime cycles.
- Per-process taint requires a per-user front-end process, since a process that has consumed Alice's private data cannot safely serve Bob without resetting its state. This has two implications. First, a process can serve at most one user session at a time. Second, after session termination, a process must be re-exec'ed to serve another session.

SHAI's design. To mitigate the sources of THOTH's overhead, SHAI uses different policy enforcement mechanisms. First, instead of intercepting I/O at runtime, SHAI relies on an *offline analysis* to certify accesses that do not violate policies. These accesses are then compiled into a set of capabilities that can be enforced directly by an operating system's *capability sandbox* at runtime. This allows SHAI to intercept I/O only sparingly, for accesses that could not be certified during the offline analysis. This design is possible since many aspects of a data retrieval system's runtime behaviour are available (or can be predicated) statically, such as normal flows of data items between the system's tasks and the policies that are in effect on those data items.

Second, instead of per-process taint, SHAI relies on operating system primitives for *in-process* isolation to support *per-session taint*. With that in place, the same process can safely serve multiple sessions concurrently. (As an added benefit, using in-process isolation allows SHAI to run its RM within the same process too. This avoids the process context switch overhead for the few accesses that could not be certified during the offline analysis and are intercepted.)

We have implemented a SHAI prototype. A background job periodically performs the offline flow analysis and can process millions of individual flows within seconds while being constrained to a single CPU core. Our prototype relies on Capsicum [89] for capability sandboxing, and on light-weight contexts [52] (*lwCs*) for efficient in-process isolation. Architecturally, a process in SHAI has an unsandboxed (privileged) monitor *lwC*, which runs SHAI's RM, and possibly many Capsicum-sandboxed (unprivileged) task *lwCs*, where each maps to a system's task or to a user's session. SHAI's RM grants access capabilities to task *lwCs* allowing accesses certified by the offline analysis. Using Apache Lucene and sufficiently long user sessions, our performance measurements indicate that SHAI's policy enforcement overhead on throughput is as little as 0.02% on a setup that achieves a few hundred requests/second/-machine. Additionally, on systems that scale up to a few tens of thousands requests/second/-machine, SHAI's overhead is only 1.2%, indicating that SHAI can maintain low overhead even in high-performance data retrieval systems.

1.3.3 How do THOTH and SHAI compare?

THOTH and SHAI are policy enforcement systems for data retrieval pipelines. Both are comprehensive: they enforce data use policies including those specific to individual data items or to a given user's data items, the provider's own policies, and policies that capture legal requirements. Both systems are practical: they incur low runtime overhead; existing applications can run with little to no modifications; and bugs, misconfigurations, or actions by unprivileged operators cannot violate policies.

Besides sharing the same goal, THOTH and SHAI share a number of design principles. Policies are specified in a declarative language, separate from application code, and directly attached to data. A policy attached to a data item is a complete and a one point description of all confidentiality and integrity requirements in effect for that data item. Policy enforcement is completely independent of application code, which could be buggy or misconfigured. THOTH and SHAI enforce policies at task boundaries. Policy enforcement at task boundaries matches the pipelined structure of data retrieval systems and makes these compliance systems independent of any language, runtime, or framework used for developing applications.

However, THOTH and SHAI differ drastically in policy enforcement techniques, representing two different design points. Each design point makes different assumptions about the information the provider is able to provide to the compliance system. THOTH's design makes no assumptions about the data flows that applications may attempt at runtime. Therefore, it relies on *dynamic* information flow control. It tracks data flows by intercepting I/O, propagates policies along these flows, and enforces policy conditions when data leaves the system or when policy changes along the data flow path. On the other hand, SHAI assumes that the provider can reasonably approximate the data flows that applications are expected to perform. It uses *offline analysis* to determine the compliance of these flows, which are then compiled into a set of fine-grained I/O capabilities that can be enforced directly by the operating system.

The different enforcement techniques of THOTH and SHAI result in different policy enforcement overheads. Our performance measurements show that SHAI's overheads can be significantly lower than THOTH's when the information available to (or predicted by) the offline analysis about the runtime behaviour of the system are accurate. In fact, our measurements indicate that SHAI's overheads are lower than THOTH's across the board, even when most of the assumptions made by the offline analysis are inaccurate.

Naming. The names for THOTH and SHAI are inspired by Egyptian mythology. Thoth is a recording angel; and Shai is the god of destiny. Dually, THOTH tracks processes' I/O; and SHAI configures sandboxes limiting what applications will be able to access.

1.4 Thesis Contributions

This thesis makes the following main contributions:

- A declarative policy language that can express the integrity and confidentiality requirements of data flows. The language can specify many individual and column-level policies that arise in practice in data retrieval systems.
- The design of two comprehensive and practical policy compliance systems, THOTH and SHAI. THOTH enforces policies by I/O interception and taint propagation, whereas SHAI relies on offline analysis and light-weight monitoring to enforce policies.
- The application of the proposed designs to a prototype search engine based on Apache Lucene.
- An optimized prototype implementation and experimental evaluation of the two systems to measure overheads.

1.5 Organization

The rest of this document is organized as follows. Chapter 2 describes background material that the systems developed in this thesis build on. Chapter 3 presents THOTH, and Chapter 4 presents SHAI. Chapter 5 discusses related work. I give concluding remarks in Chapter 6, and I outline directions for future research in Chapter 7.

2 Background

This chapter presents background on some of the tools that the work developed in this thesis uses.

2.1 Linux Security Module

Linux Security Module (LSM) [95] is a general Linux kernel framework to implement security models that rely on kernel interception of system calls. It is part of the mainstream Linux kernel since version 2.6 and has been adopted by a number of widely used security projects such as SELinux [53], AppArmor [23], Smack [85], and TOMOYO Linux [86].

LSM provides *hooks*, which are function pointers placed strategically in the kernel proper. When executing security-sensitive operations, these hooks direct the kernel control flow to a model-specific kernel module, which provides the enforcement logic. A key characteristic of LSM is that it provides comprehensive I/O interposition (i.e., complete mediation) [103]. The LSM hooks cover I/O operations (e.g., files, sockets, and shared memory) and process creation (e.g., fork and exec) among others.

Our THOTH prototype uses LSM as a basic building block for the kernel-level monitor. In particular, our implementation uses a kernel module that plugs directly into the LSM interface.

2.2 Capsicum

Capsicum [89] provides capability primitives for the UNIX family of operating systems. It is a widely used security project and is part of mainstream FreeBSD. Moreover, many popular tools, for example dhclient [29] and openSSH [66], rely on Capsicum's capability primitives to enhance security.

At a high level, Capsicum's capability primitives can be used to provide an OS-supported sandboxing mechanism. Capsicum introduces two core primitives: *capabilities* and *capability mode*. A capability is a file descriptor with associated access rights, such as read-only or read-write access. Capability mode ensures process isolation. A process in capability mode can access only process-local information, such as its existing open file descriptors. On the other hand, accesses to global namespaces (file system and process namespaces) and several management interfaces (mapping devices, and loading kernel modules) are denied. Denying access to global namespaces is critical to isolating a process, since otherwise the process could learn information about the system by querying those global namespaces (e.g., does a particular file system path exist?). A process in capability mode cannot create capabilities outside its current capability set (since creating such capabilities requires access to global namespaces).

A system can use Capsicum's primitives to enhance its resilience against program vulnerabilities. Following the principle of least privilege, a program creates (or acquires) as few capabilities as necessary to perform its intended functionality. Once the program enters capability mode, damages due to compromises are limited to what is accessible with its capability set.

Our SHAI prototype uses Capsicum for OS-supported capability sandboxing. Applications run in Capsicum sandboxes with just enough capabilities to perform accesses that were certified as policy-compliant by SHAI's offline analysis.

2.3 Light-Weight Contexts

Light-weight contexts (*lwCs*) [52] are an OS abstraction that provides independent units of protection, privilege, and execution *within a process*. A process may contain multiple *lwCs*, each with their own (or selectively shared) virtual memory mappings, file descriptor table, and credentials/capabilities. *lwCs* are orthogonal to execution threads; a thread can switch between *lwCs* through system calls. A *lwC* switch is efficient as it avoids scheduling overhead (compared to the context switch of a process/thread). The *lwC* API enables efficient compartmentalization and monitoring.

Our SHAI prototype relies on *lwCs* (coupled with Capsicum) for its runtime architecture. In SHAI, each process has an unsandboxed (privileged) monitor *lwC*, which runs SHAI's RM, and possibly many Capsicum-sandboxed (unprivileged) task *lwCs*, where each maps to an application's task or to a user's session. SHAI's RM grants access capabilities to task *lwCs* allowing accesses certified by the offline analysis. Switching between *lwCs* is efficient and incurs little overhead when invoking the monitor *lwC* to acquire capabilities. The monitor *lwC* protects the integrity of the reference monitor (by isolating it from application tasks and users' sessions), and the task *lwCs* prevent accidental leakage of private information across user sessions (by isolating them from one another). If the system calls of a task *lwC* require capabilities outside of the task's current capability set, those system calls are redirected by the underlying OS to the monitor *lwC*. **S**HAI relies on this redirection to intercept I/O to data items outside of the capability set of task *lwCs*.

3 THOTH: Policy Compliance via Runtime Monitoring

THOTH is a kernel-level policy compliance layer that helps data retrieval system providers enforce confidentiality and integrity policies on the data they collect and serve. In THOTH, the *provider* attaches policies to data items (documents and live streams, posts and profiles, user click history, etc.) based on the privacy preferences of clients, external usage requirements (e.g., legal), and internal usage requirements (e.g., audit). The policy attached to a data item is a *complete, one point description* of all privacy and integrity rules in effect for that data item. THOTH *tracks data flows* by intercepting all IPC and I/O in the kernel, and it propagates policies along these flows. It enforces policy when data leaves the system or when policy changes along the data flow, regardless of bugs, misconfigurations, or errors by unprivileged operators. We next briefly describe the key insights in THOTH's design.

Policies separate from application code. A policy specifying *all* the confidentiality and integrity requirements may be associated with any data conduit, i.e, a file, key-value tuple, named pipe or network connection, and is enforced on all application code that accesses the conduit's data or data derived from that data. THOTH provides a declarative language for specifying policies. The language itself is novel; in addition to standard access (read/write) policies, it also allows specifying data declassification policies by stipulating how access policies may change along a data flow.

Coarse-grained dynamic analysis. As we have mentioned earlier, individual policies may not be amenable to static analysis (precision loss, policies refer to information available only at runtime). Hence, THOTH uses coarse-grained dynamic analysis. It intercepts I/O in the kernel, tracks the flow of data at the granularity of conduits and processes, and enforces policies at process boundaries. This incurs a runtime overhead but we show that the overhead is low. With an optimized prototype implementation applied to a search pipeline based on Apache

Lucene, we measure an overhead of 0.7% on indexing, and an overhead of 3.6% on query throughput under a few hundred requests/second/machine. While this overhead may be too high for large-scale data retrieval systems that need to sustain higher request rates, we believe that it is suitable for many domain-specific data retrieval systems run by organizations, enterprises and governments. Moreover, application code requires very few changes to run with THOTH (50 lines in a codebase of 300,000 LoC in our experiments).

Novel declassification policy constructs. The complexity of a data retrieval system often necessitates some declassification to maintain functionality. For instance, an index computed over a corpus of the private data of more than one individual will have a policy that disallows any of those individuals from reading. (Otherwise, one individual may read the private data of another). Therefore, a search process that consults such index cannot produce any readable results without declassification. To handle this and similar situations, we introduce a new form of declassification called *typed declassification*, which allows the declassification of data in specific forms (types). To accommodate the aforementioned search process, all source data policies allow declassification into a list of search results (document names). Hence, the search process can function as usual. At the same time, the possibility of data leaks is limited to a very narrow channel: To leak information from a private file, the search process' code must maliciously encode the information in a list of valid document names. Given that the provider has a genuine interest in preventing data breaches and that the search process is an internal component that is unlikely to be compromised in a casual external attack, the chance of having such malicious code in the search process is low. Note that typed declassification needs content-dependent policies, which our policy language supports.

To summarize, the contributions of THOTH are:

- A policy language that can express individual access and declassification policies declaratively (Subsection 3.1.2 and Section 3.2).
- The design of a kernel-level monitor to enforce policies by I/O interception and lightweight taint propagation (Section 3.1).
- Application of the design to a data retrieval systems that serves more than 300 search requests/second/machine based on Apache Lucene (Section 3.4).
- An optimized prototype implementation and experimental evaluation to measure overheads (Section 3.3 and Section 3.5).

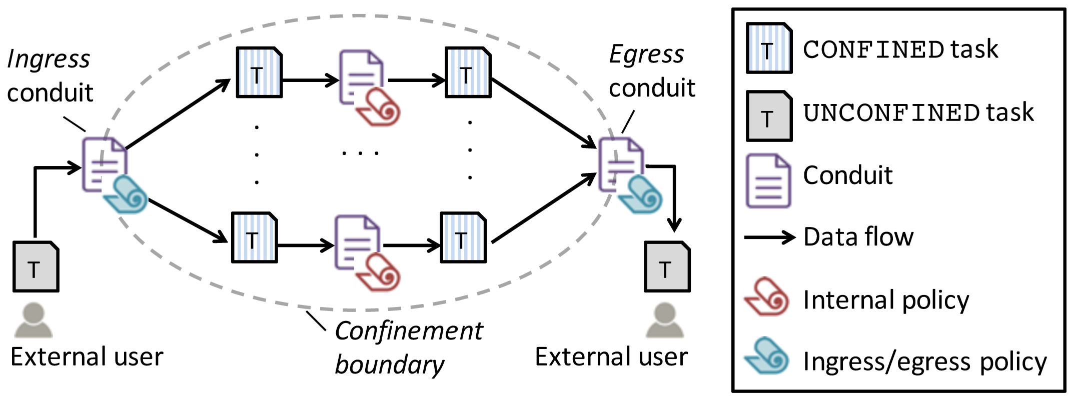


FIGURE 3.1: THOTH data flow.

3.1 THOTH Design and Architecture

THOTH is a policy compliance system that helps data retrieval system providers enforce confidentiality and integrity policies on the data they collect and serve. We next describe THOTH's data flow model, policy language, overall architecture, and threat model.

3.1.1 Data flow model

Figure 3.1 shows the data flow model of a THOTH-protected system. An application consists of a set of tasks (i.e., processes), and data flows among tasks via *conduits*. A file, a named pipe or a tuple in a key-value store is a conduit. A network connection or a named pipe is a pair of conduits, one for each direction of data traffic. THOTH identifies each conduit with a unique numeric identifier, called the conduit id. The conduit id is the hash of the path name in case of a file or named pipe, the hash of the 5-tuple ⟨srcIP, srcPort, protocol, destIP, destPort⟩ in case of a network connection, or the key in case of a key-value tuple. Any conduit may have an associated policy.[1]

The core of the application system is a set of CONFINED tasks within THOTH's *confinement boundary*. The system interacts with the outside world via conduits (typically network connections) to external, UNCONFINED tasks. An UNCONFINED task represents an external user (or an external component) and may possess the user's authentication credentials. Neither type of tasks is trusted by THOTH.

[1]If a file has multiple hard links, each of its path names can be associated with a different policy. When a path name is used to access the file, that path name's policies are checked.

Policies on inbound and outbound conduits that cross the confinement boundary represent the ingress and egress policies, respectively. These ingress and egress policies collectively control how data can be used within the system, and how the data can be disseminated (and fed) from (and into) the system.

3.1.2 Policy language design

THOTH policies are specified in a new, expressive declarative language, separate from application code. A policy can be attached to any conduit. The policy on a conduit protects the confidentiality and integrity of the data in the conduit. THOTH policies are specified in two layers.

The first layer, an *access control policy*, specifies which principals may **read** and **update** the conduit and under what conditions (e.g., only before or only after a certain date). This layer has a read rule and an update rule. Both rules are written in the syntax of Datalog, which has been used widely in the past for the declarative specification of access policies [16, 28, 51]. Briefly, the read rule has the form (**read** :- cond) and means that the conduit can be read if the condition "cond" is satisfied. The condition "cond" consists of *predicates* connected with conjunction ("and", written $\wedge$) and disjunction ("or", written $\vee$). Similarly, the update rule has the form (**update** :- cond).

The second layer protects data *derived* from the conduit by restricting the policies of all *downstream sequences of conduits* in the data pipeline. This layer can **declassify** data by allowing the access policies downstream to be relaxed progressively, as more and more declassification conditions are met. This layer contains a single rule and has the form (**declassify** :- cond), where "cond" is a condition or predicate on all downstream sequences of conduits. For instance, "cond" may say that in any downstream sequence of conduits, the access policies must allow read access only to Alice, until the calendar year is at least 2020, after which the policies may allow read access to anyone. This represents the declassification policy "private to Alice until 2020". We represent such declassification policies using the notation of *linear temporal logic* (LTL), which provides a well-known operator to represent predicates that change over time [55]. We allow a new connective in "cond" in the **declassify** rule: c1 until c2, which means that condition c1 must hold of all downstream conduits until condition c2 holds. Also, we allow a new predicate isAsRestrictive(p1, p2), which checks that policy p1 is at least as restrictive as p2. The second layer that specifies declassification by controlling downstream policies is the language's key novelty. Another noteworthy feature is that we allow policy evaluation to depend on a conduit's state—both its data and its metadata (e.g., policy).

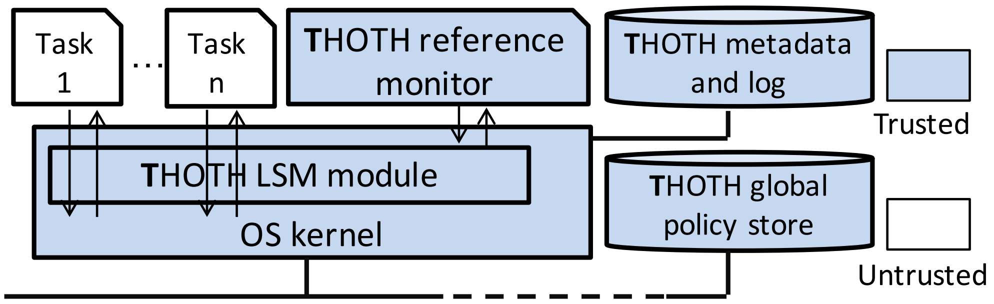


FIGURE 3.2: THOTH architecture.

This allows expressing content-dependent policies and, in particular, a kind of declassification that we call *typed declassification*. Typed declassification permits limited information flows in specific forms (types).

In the context of THOTH's data flow model, the read and declassification rules of an ingress policy control how data can be used and disseminated by the system, whereas the update rule of an ingress policy determines who may feed data into the system. The read rule of an egress policy defines who outside the system could read the output data.

3.1.3 Architecture

THOTH is a distributed policy enforcement layer that ensures compliance via *dynamic coarse-grained flow control*. Figure 3.2 depicts THOTH's architecture. At each participating node, THOTH comprises a kernel module, a trusted reference monitor process, a persistent store for metadata and transaction log, and a persistent policy store. THOTH maps applications' tasks to processes. The kernel module intercepts I/O at the process boundary and re-directs it to the reference monitor. The reference monitor evaluates policies and maintains per-task taint sets. A task's taint set is the set of policies of all the conduits it has consumed in the past. The reference monitor has exclusive access to the policy store, which provides a consistent view of all policies. Such consistent view can be attained by using either centralized storage for policies or via a consensus protocol (such as Paxos [50]).

Dynamic coarse-grained flow control justification. At an abstract level, enforcing THOTH's policies requires determining, for each egress conduit, which ingress conduits' data could flow to it, and what declassification conditions (if any) must be satisfied along the flow. This is a standard data flow analysis problem, for which many different techniques have been proposed in the literature. We briefly outline these existing techniques and their shortcomings in the context of (**T**HOTH-like) data-specific policies.

Static techniques determine flows by analyzing the source code of the system [27, 2]. In addition to requiring the source code and being language-specific, static techniques (as we explained earlier) cannot enforce individual, data-specific policies, and therefore, are not suitable for **T**HOTH.

Dynamic fine-grained techniques, also known as runtime taint tracking techniques, track data flows between program variables, or between memory objects and machine registers at runtime [74, 98]. Depending on the specific implementation, a dynamic fine-grained technique may not have the shortcomings of static techniques mentioned above, but dynamic fine-grained techniques must intercept all memory and register reads and writes. This interception makes their overhead prohibitively high for most online systems (in the orders of upper 10s to 100s of percent).

Dynamic coarse-grained techniques track flows at coarser granularity, typically only across tasks in a system but not within each task [31, 101, 49]. They only intercept reads and writes at task boundaries. This is far more efficient than tracking all reads and writes to registers and memory. Theoretically, this comes at the cost of precision—if a task reads a conduit f and later writes a conduit g, a coarse-grained technique must conservatively assume that there is a flow from f to g, even if the data written to g was independent of the data read from f. Admittedly, such loss of precision can cause overtainting. Yet, we argue next that, despite the potential risk of overtainting, dynamic coarse-grained tracking is a reasonable option for enforcing data use policies in a data retrieval system. We distinguish between two causes for overtainting:

1. Overtainting can be an artifact of a given system implementation. Data subject to different policies may *accidentally* intermix within a task, where the intermixing of data is not fundamental to the computation. For instance, a web server implementation that uses the same task to serve multiple clients falls into this category. Mitigating such overtainting can be achieved via a cautious implementation that avoids the unnecessary intermixing of flows. For example, another implementation of the web server that uses a separate task per client will mitigate that kind of overtainting.

2. Overtainting can be inherent to the computation itself where a task's output depends on multiple data sources that are subject to different policies. Consider, for example, a search engine that computes a single index over a large data corpus (public web pages, corporate documents, users' data, etc.). Intermixing data in order to generate the index

is inherent to the index computation, as the quality of search results (i.e., relevance) improves with a larger index that contains more documents [42, 97]. Traditionally, dynamic coarse-grained tracking systems use trusted application components to declassify output in the presence of overtainting [31, 101, 49]. In THOTH, the declassification requirements are encoded in the policies (as applications are assumed to be buggy). Therefore, working around this kind of overtainting is possible only when the involved data use policies can be (slightly) relaxed to allow legitimate data flows.

In the context of data retrieval systems, our practical experience suggests the following: First, data retrieval systems can avoid the unnecessary, accidental intermixing of data by using a cautious implementation that factors the computation into tasks such that no computationally independent data flows are mixed within a task. In fact, many distributed computing frameworks, such as MapReduce [25], allow computations to be factored in that manner. Second, a slight relaxation of policies is possible to overcome overtainting inherent to the computation of data retrieval systems when it arises. (We give an example of such relaxation in 3.2.2). Consequently, dynamic coarse-grained tracking is a reasonable option for enforcing data use policies in a data retrieval system structured as a pipeline of tasks. We provide details of a prototype data retrieval system where we apply coarse-grained tracking in Section 3.4.

3.1.4 Threat model

THOTH's goal is to prevent inadvertent data leaks due to application bugs, misconfigurations, and errors by unprivileged operators. The THOTH kernel module and reference monitor, as well as the Linux system and policy store they depend on, are trusted. Leaks due to vulnerabilities in these components are out of scope. We assume that correct policies are installed on ingress and egress conduits. In our current prototype, storage systems that hold application data are assumed to be trusted. This assumption can be relaxed by encrypting and checksumming application data in the THOTH kernel module.

THOTH makes no assumptions about the nature of bugs and misconfigurations in application components, the type of errors committed by unprivileged operators, or errors in policies on internal conduits. Subject to this threat model, THOTH provably enforces all ingress policies. In information flow control terms, THOTH can control both explicit and implicit flows, but leaks due to malicious adversaries, covert, and side-channels are out of scope.

Threat model justification

Trust assumptions. Trusting the THOTH kernel module, reference monitor, and the Linux system they depend on is reasonable in practice because *(i)* reputable providers will install security patches on the OS and THOTH components, and install correct policies; and *(ii)* the OS and THOTH are maintained by a small team of experts and are more stable than the applications that are maintained by significantly larger number of engineers and evolve more rapidly.

Malicious adversaries. As we mentioned earlier, data leaks due to malicious adversaries are out of scope. Nonetheless, THOTH prevents direct flows to unauthorized parties, and can, therefore, protect against *some* malicious applications (and active attacks on vulnerable applications) that rely on such direct flows. THOTH does not protect against malicious applications that leak data through side-channels. Moreover, typed declassification policies admit limited information flows, which can be exploited by malicious applications covertly. For instance, a search process that consults an index computed over a corpus containing the private data of more than one individual cannot produce any readable results without declassification. If source data policies allow declassification into a list of conduit ids, the search process can then function as usual. However, malware injected into the search engine can encode private information in the set of conduit ids it produces. This channel is out of scope. In practice, such attacks require significant sophistication. A successful attack must inject code strategically into the data flow before a declassification point and encode private data on a policy-compliant flow.

To summarize, THOTH prevents accidental policy violations due to application bugs, misconfigurations, and errors by unprivileged operators. (As an added benefit from THOTH's data flow control, THOTH prevents active attacks that rely on direct flows to unauthorized parties). We demonstrate THOTH's ability to prevent data leaks in Subsection 3.5.3 where a THOTH compliant search engine is able to enforce data policies, preventing (real and synthetic) bugs and misconfigurations from leaking information.

3.1.5 Data flow tracking and enforcement

Tracking data flow

THOTH tracks data flows coarsely at the task-level (i.e., process boundaries). `CONFINED` and `UNCONFINED` tasks are subject to different policy checks. A `CONFINED` task may read any conduit, irrespective of the conduit's **read** rule, but THOTH enforces each such conduit's **declassify** rule when the task writes to other conduits. To do this, **T**HOTH maintains the **declassify** rules of conduits read by each CONFINED task in the task's metadata (these rules constitute the *taint set* of the task).

UNCONFINED tasks form the ingress and egress points for **T**HOTH's flow tracking; they are subject to access control checks, not tainting. An UNCONFINED task may read from (write to) a conduit only if the conduit's **read** (**update**) rule is satisfied. For example, to read Alice's private data, an UNCONFINED task must authenticate with Alice's credentials. Conduits without policies can be read and written by all tasks freely.

In summary, **T**HOTH tracks data flows across CONFINED tasks coarsely, and enforces declassification policies on these flows. At the ingress and egress tasks (UNCONFINED tasks), **T**HOTH imposes access control through the **read** and **update** rules. Every new task starts UNCONFINED. The task may transition to the CONFINED state through a designated **T**HOTH API call. The reverse transition is disallowed to prevent a task from reading private data in the CONFINED state and leaking the data to a conduit without any policy protection after transitioning to the UNCONFINED state.

Conduit interceptors

The **T**HOTH kernel component includes a conduit interceptor (CI) for each type of conduit. A CI for a given conduit type intercepts system calls that access or manipulate conduits of that type, and associates a conduit with its policy. **T**HOTH has built-in CIs for kernel-defined conduit types, namely files, named pipes, and network connections. CIs for additional conduit types can be plugged in. For instance, our prototype uses a CI for the memcached key-value store (KV).

The CIs for files and named pipes associate a policy with the unique pathname of a file or pipe. The socket CI associates a policy with the network connection's 5-tuple ⟨srcIP, srcPort, protocol, destIP, destPort⟩. The 5-tuple may be underspecified. For instance, the policy associated with ⟨?, ?, ?, destIP, destPort⟩ applies to any network connection with the specified destination IP address and port. Both ends of a network connection have the same policy. The KV CI associates a policy with a tuple's key. The KV CI can automatically derive policies from policy templates that cover a subspace of keys (e.g., all keys with prefix $\#user\_profile$). It can also replace template variables with metadata, e.g., the time at which the key was created.

Algorithm 1 THOTH policy enforcement algorithm.

```
Inputs:   t, the task reading or writing the conduit
          f, the conduit being read or written
          op, the operation being performed (read or write)
Output: Allow or deny the access
Side-effects: May update the taint set of t

if t is UNCONFINED:
  if op is read:
    Check f's **read** rule.
  if op is write:
    Check f's **update** rule.

if t is CONFINED:
  if op is read:
    Add f's policy to t's taint set.
  if op is write:
    // Enforce declassification policies of t's taint set
    for each **declassification** rule (c until c') in t's taint set:
      Check that EITHER c' holds OR (c holds AND
      f's declassification policy implies (c until c')).
```

Policy enforcement algorithm

Algorithm 1 summarizes the abstract checks that **T**HOTH makes when it intercepts a conduit access. If the calling task is UNCONFINED, then **T**HOTH evaluates the read or update policy of the conduit (lines 1–5). If the calling task is CONFINED and the operation is a read, then **T**HOTH adds the policy of the conduit being read to the taint set of the calling task. No policy check is performed in this case (lines 6–8). To reduce the size of a CONFINED task's taint set, our prototype performs *taint compression* when possible: A policy is not added if the taint set already includes an equally or more restrictive policy.

When a CONFINED task t writes a conduit f, there is a potential data flow from every conduit that t has read in the past to f. Hence, all declassification rules in t's taint set are enforced (lines 11–12). Suppose (c until c′) is a **declassification** rule in t's taint set. Since this rule means that condition c must continue to hold downstream *until* the declassification condition c′ holds, this rule can be satisfied in one of two ways: Either the declassification condition c′ holds now, or c holds now and the next downstream conduit (f here) continues to enforce (c until c′). Line 12 makes exactly this check.

End-to-end correctness of policy enforcement. Within the threat model of THOTH, the checks described above enforce all policies on conduits and, specifically, all ingress policies. Incorrect policy configuration on internal conduits cannot cause violation of ingress policies but may cause compliant data flows to be denied by the THOTH reference monitor. Informally, this holds because our checks ensure that the conditions in every declassification policy are propagated downstream until they are satisfied.

Policy comparison. THOTH compares policies for restrictiveness in three cases: for taint compression, when evaluating the predicate isAsRestrictive(), and in line 12 of the enforcement algorithm (Algorithm 1). The general comparison problem is undecidable for first-order logic, so THOTH uses the following heuristics:

(1) Equality: Compare the hashes of the two policies. Two policies are equally restrictive when their hashes match (syntactic equality).

(2) Inclusion: Check that all predicates in the less restrictive policy also appear in the more restrictive one, taking into account variable renaming and conjunctions and disjunctions between the predicates. Inclusion has exponential time complexity in the worst case, but is fast in practice.

(3) Partial evaluation: Evaluate and delete an application-specified subpart of each policy, then try equality and inclusion.

These heuristics suffice in all cases we have encountered. Note that a policy comparison failure can never affect THOTH's safety. However, a failure can (a) defeat taint compression and therefore increase taint size and policy evaluation overhead; or (b) cause a compliant data flow to be denied. In the latter case, a policy designer may re-state a policy so that the policy comparison succeeds.

3.1.6 THOTH API

Table 3.1 lists THOTH API calls. User-level tasks can invoke these calls through a system call that THOTH introduces. The THOTH API allows tasks to transition from the UNCONFINED to the CONFINED state using *confine()*, to authenticate directly with THOTH's reference monitor with a private key using *authenticate()*, and to set/get policies on conduits using *add_policy(), set_policy(), or get_policy()*.

API call	**Description**
confine ()	Transition the calling process from UNCONFINED to CONFINED state.
authenticate (k)	Authenticate a process with the private key *k*. A process must authenticate to be able to satisfy identity-based policies.
add_policy (p)	Store a policy *p* in THOTH metadata, create and return a policy id *p_id* for *p*.
set_tx_flags (c_id, flags)	Set flags *flags*, such as partial evaluation hints, for a transaction on conduit *c_id*.
open_tx (c_id)	Open a transaction on conduit *c_id* and return a file handle *fd*.
close_tx (fd)	Close a transaction *fd*. Return 0 if successful, or error code if a policy check fails.
set_policy (fd, p_id)	Attach policy id *p_id* to the conduit running transaction *fd*. Passing (-1) for *p_id* sets the null policy. The new policy is applied only after *fd* is successfully closed. The declassification condition of the conduit's existing policy determines whether the policy change or removal is allowed.
get_policy (c_id, buf)	Retrieve the policy attached to conduit *c_id* into buffer *buf*.
cache (fd, off, len)	Cache content (for policy evaluation) from file handle *fd* from offset *off* with length *len*.

TABLE 3.1: THOTH API calls.

Transactions. To check structural properties of written data (e.g., that the data is a list of conduit ids), it is often necessary to evaluate the **update** rule atomically on a *batch* of writes. Hence, THOTH supports write transactions on conduits. By default, a transaction starts with a POSIX open() call and ends with the close() call on a conduit. Transactions can also be explicitly started and ended using the THOTH API calls *open_tx()* and *close_tx()*.

Tasks can pass hints to THOTH to improve the efficiency of policy evaluation during transactions. Such hints are not trusted and do not affect the correctness of policy evaluation. For instance, tasks can mark which policy parts should be subject to partial policy evaluation using *set_tx_flags()*, or cache conduit content relevant to policy evaluation using *cache()*.

During a transaction, THOTH buffers writes in a persistent re-do log. When the transaction is closed by the application, THOTH makes the policy checks described in Algorithm 1. If the policy checks succeed, then the writes are sent to the conduit, else the writes are discarded. The re-do log allows recovery from crashes and avoids expensive file system syncs when a transaction commits. As an optimization, THOTH allows writes to be sent directly to the conduit during a transaction, skipping buffering in the re-do log, when the conduit is not subject to an integrity policy (i.e., **update** :- $TRUE$).

3.1.7 Summary

THOTH enforces conduits' policies despite application-level bugs, misconfigurations, and actions by unprivileged operators. A data source's policy specifies both access and declassification conditions and completely describes the allowed usages of the source. THOTH uses policies as taint, which differs significantly from the standard information flow control practice of using abstract labels as taint [26, 31, 101, 49]. That practice requires trusted application processes to declassify data and to control access at system edges. In contrast, THOTH relies entirely on its reference monitor for all access and declassification checks, and no application processes have to be trusted.

3.2 Example Policies

In this section, we discuss example policies that clients, data sources, and the provider might wish to enforce in a data retrieval system, and give a glimpse of how to express these policies in THOTH's policy language. All supported predicates in the policy language are listed in Table 3.2.

3.2.1 Client policies

We start with client policies that capture the preferences of data owners.

Private policies

Consider a search engine that indexes clients' private data. A relevant security goal might be that a client Alice's private e-mails and profile should be visible only to Alice, and only

Arithmetic/String		**Session**	
add(x,y,z)	x=y+z	sKeyIs(x)	x is the session's authentication key
sub(x,y,z)	x=y-z		
mul(x,y,z)	x=y*z	sIpIs(x)	x is the session's source IP address
div(x,y,z)	x=y/z		
rem(x,y,z)	x=y%z	IpPrefix(x,y)	x is IP prefix of y
concat(x,y)	x || y	timeIs(t)	t is the current time
vType(x, y)	is x of type y?		
Relational		**Conduit**	
eq(x,y)	x=y	cNameIs(x)	x is the conduit pathname
neq(x,y)	x!=y	cIdIs(x)	x is the conduit id
lt(x,y)	x<y	cIdExists(x)	x is a valid conduit id
gt(x,y)	x>y	cCurrLenIs(x)	x is the conduit length
le(x,y)	x<=y	cNewLenIs(x)	x is the new conduit length
ge(x,y)	x>=y	hasPol(c, p)	p is conduit c's policy
		cIsIntrinsic	does this conduit connect two confined processes?

Content	
(c,off) says $(x_1, .., x_n)$	tuple $x_1, .., x_n$ is in conduit c at off
(c,off) willsay $(x_1, .., x_n)$	ditto for of c's update transaction
each in (c,off) says $(x_1, .., x_n)$ {cond}	for each tuple in c at off: assign to x_1,.., x_n & evaluate condition
each in (c,off) willsay $(x_1, .., x_n)$ {cond}	ditto for c's update transaction
sListIncludes (c,$(x_1, .., x_n)$)	c contains $x_1, .., x_n$ (c must be sorted)
sListExcludes (c,$(x_1, .., x_n)$)	c excludes $x_1, .., x_n$ (c must be sorted)

Declassification rules	
c1 until c2	condition c1 must hold on the downstream flow until c2 holds
isAsRestrictive(p1,p2)	the permission p1 is at least as restrictive as p2

TABLE 3.2: **THOTH** policy language predicates and connectives.

she should be able to modify this data. This *private data* policy can be expressed by attaching to each conduit holding Alice's private items **read** and **update** rules that allow these operations only in the context of a session authenticated with Alice's key. The latter condition can be expressed using a single predicate sKeyIs($k_{\texttt{Alice}}$), which means that the active session is authenticated with Alice's public key, here denoted $k_{\texttt{Alice}}$. Hence, the read rule would be **read** :- sKeyIs($k_{\texttt{Alice}}$). The update rule would be **update** :- sKeyIs($k_{\texttt{Alice}}$). (Clients, or processes running on behalf of clients, authenticate directly to **THOTH**, so **THOTH** does not rely on untrusted applications for session authentication information.)

Friends-only policies

Alice's *friends-only* blog and online social network profile should be readable by Alice and her friends, which can be expressed with an additional disjunctive clause in the read rule:

read :- sKeyIs($k_{\texttt{Alice}}$) $\vee$
(sKeyIs(K) $\wedge$ ("Alice.acl", `Offset`) says isFriend(K))

The part after the $\vee$ is read as "the key K that authenticated the current session exists in Alice.acl at some offset `Offset`." Here, Alice.acl is a key-value tuple that contains Alice's friend list.

Following standard Datalog convention, terms like K and `Offset` that start with uppercase letters are existentially quantified variables. The predicate sKeyIs(K) binds K to the key that authenticates the session. During each policy evaluation, application code is expected to provide a binding for the variable `Offset` that refers to a location in the tuple's value saying that K belongs to a friend of Alice. Note that policy compliance does not depend on application correctness: access is denied if the application does not provide a correct offset.

Friends-of-friends policies

Extending further, visibility to Alice's *friends of friends* can be allowed by modifying the read rule to check that Alice and the owner of the current session's key have a common friend. Then, the application code would be expected to provide an offset in Alice's ACL where the common friend exists and an offset in the common friend's ACL where the current session's key exists.

3.2.2 Index policy

In the previous example, we discussed confidentiality policies that reflect data owners' privacy choices. For the retrieval system to do its job, however, the input data policies must allow some declassification. The search pipeline is one instance where declassification is needed. Consider an index that is computed over the entire corpus, including the private data of several individuals. No individual should be able to read such an index (otherwise the system risks leaking the private data of one individual to another). Consequently, the search engine, which consults such an index would not be allowed to produce any output to any user. In order to allow the search engine to produce readable output, *all* the data sources that are used to create the index must allow some declassification.

One declassification that allows the search engine to produce readable output (and allows the search pipeline to function properly) is *declassifying the search results* (i.e., pathnames). We rely on the policy language's novel ability to refer to a conduit's (meta-)data to allow the selective, *typed declassification* of the search engine's output. The declassification policy can be implemented by adding the following **declassify** rule to all searchable input data:

$$\begin{aligned}\textbf{declassify} \ :\!- \ & \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read}) \ \mathsf{until} \\ & \text{ONLY\_CND\_IDS}\end{aligned}$$

When this policy is set on a conduit c, data derived from c can be written into conduits whose read rule is at least as restrictive as c's **read** rule (which is bound to `this`.**read**), until it is written into a conduit which satisfies the condition ONLY_CND_IDS. This macro stipulates that only a list of valid conduit ids has been written. The macro expands to

$$\begin{aligned}& \mathsf{cCurrLenIs}(\texttt{CurrLen}) \land \mathsf{cNewLenIs}(\texttt{NewLen}) \land \\ & \mathsf{each\ in}(\mathsf{this}, \texttt{CurrLen}, \texttt{NewLen}) \ \mathsf{says}(\texttt{CndId}) \\ & \quad \{\mathsf{cIdExists}(\texttt{CndId})\}\end{aligned}$$

and permits the declassification of a list of proper conduit ids. The predicate "each in () says () {}" iterates over the sequence of tuples in the newly written data and checks that each is a valid conduit id. By including this declassification rule in her data item's policy, Alice allows the search engine to index her item and to include it in search results. To view the contents, of course, a querier still has to satisfy each conduit's confidentiality policy.

Confidential conduit ids. So far, we have assumed that the conduit ids (i.e., the pathnames of indexed files) are not themselves confidential. If the conduit ids are themselves confidential, then the above **declassify** rule is insufficient since it stipulates no restriction on policies after ONLY_CND_IDS holds. Thus, a more restrictive **declassify** rule is needed. Ideally, we want that the read and declassify rules of the conduit that contains the list of conduit ids be at least as restrictive as the read and declassify rules of all conduits in the list. This can be accomplished by the following replacement for ONLY_CND_IDS:

$$\begin{array}{l}\mathsf{cCurrLenIs}(\texttt{CurrLen}) \wedge \mathsf{cNewLenIs}(\texttt{NewLen}) \wedge \\ \mathsf{each\ in}(\mathsf{this}, \texttt{CurrLen}, \texttt{NewLen})\ \mathsf{willsay}(\texttt{CndId}) \\ \{\mathsf{cIdExists}(\texttt{CndId}) \wedge \mathsf{hasPol}(\texttt{CndId}\ , \texttt{P}) \wedge \\ \ \mathsf{isAsRestrictive}(\mathbf{read}, \texttt{P}.\mathbf{read}) \wedge \\ \ \mathsf{isAsRestrictive}(\mathbf{declassify}, \texttt{P}.\mathbf{declassify})\}\end{array}$$

The predicate hasPol(`CndId`, `P`) binds `P` to the policy of the conduit `CndId`, and the predicates isAsRestrictive(**read**, `P`.**read**) and isAsRestrictive(**declassify**, `P`.**declassify**) enforce that the read and declassify rules of the search results are at least as restrictive as those of `CndId`. We call this modified macro ONLY_CND_IDS+.

Hence, when the conduit ids themselves are confidential, the searchable input data should have the **declassify** rule:

$$\mathbf{declassify}\ \text{:-}\ \mathsf{isAsRestrictive}(\mathbf{read}, \texttt{this}.\mathbf{read})\ \mathsf{until}\ \text{ONLY\_CND\_IDS+}$$

With this declassify rule in place, the search engine can write the search results, following the form (`CndId1`, `CndId2`, ..., `CndIdn`), only to a conduit whose **read** and **declassify** rules are at least as restrictive as all the **read** and the **declassify** rules, respectively, of the conduits referenced by (`CndId1`, `CndId2`, ..., `CndIdn`).

3.2.3 Other data retrieval policies

We briefly describe several other policies relevant to data retrieval systems that we have represented in our policy language and implemented in our prototype. For the formal encodings of these policies, see Appendix A.

Data analytics. Many retrieval systems transform logs of user activity into a user preferences vector, which is used for targeting ads, computing user profiles, and providing recommendations. Raw logs of user clicks and queries are typically private, so a profile vector derived from them cannot be used for any of these purposes without a declassification. A policy that allows typed declassification into a vector of a fixed size can be attached to raw user logs to ensure that the raw logs cannot be leaked from the system, but that the profile vector can be used for the above-mentioned purposes.

Provider policies. The provider may need to censor certain documents when a query arrives from a particular country. For this purpose, the system uses a map of IP address prefixes to countries. Separately, the provider maintains a per-country blacklist, containing a list of censored conduit ids. The *censorship policy* takes the form of a common declassification rule on source files. The rule requires that, at a conduit connecting to a client, the client's IP prefix is looked up in the prefix map, and the corresponding blacklist is checked to see if any of the search results are censored. Both the prefix map and the blacklist are maintained in sorted order for efficient lookup. The sort order is enforced by an integrity policy on the blacklists and the prefix map.

A second common provider policy allows employees to access client's private data for troubleshooting purposes, as long as such accesses are logged for auditing. A *mandatory access logging (MAL)* policy can be added for this purpose. The policy allows accesses by authorized employees, if and only if an entry exists in a separate log file, which includes a signature by the employee, the conduit being accessed, and a timestamp. The log file itself has an integrity policy that allows appends only, thus ensuring that an entry cannot be removed or overwritten. Finally, data sources must consent to provider access by allowing declassification into a conduit readable by authorized employees subject to MAL.

3.3 THOTH Prototype

Our prototype consists of a Linux kernel module that plugs into the Linux Security Module (LSM) interface, and a reference monitor. We also changed a few (22) lines of the kernel proper to provide additional system call interception hooks not included in the LSM interface, and a new system call that allows applications to interact with THOTH. A small application library consisting of 840 LoC exports the API calls shown in Table 3.1 based on this system call.

3.3.1 LSM module

The THOTH LSM module comprises approximately 3500 LoC and intercepts I/O related system calls including open, close, read, write, socket, mknod, mmap, etc. Intercepted system calls are redirected to the reference monitor for taint tracking and validation. The module includes conduit interceptors for files, named pipes and sockets, as well as interceptors for client connections to a memcached key-value store [59].

3.3.2 THOTH reference monitor

THOTH's reference monitor is implemented as a trusted, privileged userspace process. It implements the policy enforcement logic and maintains the process taint, session state and transaction state in DRAM. The monitor accesses the persistent THOTH metadata store, which includes per-conduit metadata (conduit pathname, conduit id, a pointer to the policy in effect in the policy store, and for each persistent file conduit, its current size), the transaction log, and the global policy store. The metadata and transaction log are stored in NVRAM. A write-through DRAM cache holds recently accessed metadata and policies.

The monitor is multi-threaded so it can exploit multi-core parallelism. Each worker thread invokes the THOTH system call and normally blocks in the LSM module waiting for work. When an application issues a system call that requires an action by the reference monitor, a worker thread is unblocked and returns to the reference monitor with appropriate parameters; when the work is done, the thread invokes the system call again with the appropriate results causing the original application call to either be resumed or terminated. As an optimization, the LSM seeks to amortize the cost of IPC by buffering and dispatching multiple asynchronous requests to a worker thread whenever possible. The reference monitor was implemented in 19,000 LoC of C, not counting the OpenSSL library used for secure sessions and cryptographic operations.

3.3.3 Prototype limitations

There are few missing features in our current THOTH prototype.

- Interception is not yet implemented for all I/O-related system calls.
- Memory-mapped files are currently only supported with RO access (i.e., read-only). Supporting RW access (i.e., read-write) for memory-mapped files is straightforward for files whose integrity policies are not content-dependent. To fully support RW access for memory-mapped files (including files with content-dependent policies), THOTH must prevent syncing to the underlying filesystem until a successful close() to check integrity. This requires engineering effort to expose THOTH's re-do log to the kernel's page cache,

where the page cache would flush the memory-mapped file writes to THOTH's re-do log instead of the underlying filesystem (pending a successful close()).

None of these missing features are used by our prototype data retrieval system.

3.4 Policy-Compliant Data Retrieval with THOTH

We use THOTH for policy compliance in a data retrieval system built around a distributed Apache Lucene search engine. While Apache Lucene's architecture is not appropriate for large, public search engines like Google or Bing, it is frequently used in smaller, domain-specific data retrieval systems.

3.4.1 Baseline configuration

We first describe the baseline configuration without THOTH.

Search engine. Apache Lucene is an open-source search engine written in Java [3]. It consists of an indexer and a search component. The sequential indexer is a single process that scans a corpus of documents and produces a set of index files. The search component consists of a multi-threaded process that executes search queries in parallel and produces a set of corpus file names relevant to a given search query. The size of the Apache Lucene codebase is about 300,000 LoC.

Lucene can be configured with replicated search processes to scale its throughput. Here, multiple nodes run a copy of the search component, each with the full index. A search query can be processed by any machine. Lucene can also be sharded to scale with respect to the corpus size. In this case, the corpus is partitioned, each partition is indexed individually, and multiple nodes run a copy of the search component, each with one partition index. A search query is sent to all search components, and the results combined. Replication and sharding can be combined in the obvious way.

Front-end processes. A simple front-end process accepts user requests from a remote client and forwards search queries to one or more search process(es) via a pipe. The search process(es) may forward the query to other search processes with disjoint shards. When the front-end receives the search results (a list of document file names), it produces a HTML page with a URL and a content snippet from each of the result documents, and returns the page to the Web client. When the client clicks on one of the URLs, the front-end serves the content.

A second, simple account manager front-end process accepts connections from clients for the purpose of creating accounts, managing personal profiles and policies. Clients choose from a set of policy templates for documents they have contributed to the corpus, and for their personal profile information and activity history.

Search personalization and advertising. To include typical features of a data retrieval system, we added personalized search and targeted advertising components. Since developing methods for personalized search or advertising is out of this thesis's scope, we include only basic components to model the data flows of these subsystems.

A memcached daemon runs on each search node to provide a distributed key-value store for per-user information, including a suffix of the search and click histories, profile information, and the public key. The front-end process appends a user's search queries and clicks to the histories. It uses the profile information to rewrite search queries, re-order search results, and select ads for inclusion in the results page.

An aggregator process periodically analyses a user's search and click history, and updates the personal profile information accordingly. We are not really concerned with the details of user profiling, personalized search, or ad targeting. It suffices for our purposes to capture the appropriate data flows.

3.4.2 Controlling data flow with THOTH

THOTH ensures compliance with ingress and egress policies regardless of the applications running in the system. Nonetheless, the provider may need to make minor modifications in order to permit legitimate flows. In this section, we describe the changes we introduced to the baseline configuration of the Apache Lucene search engine in THOTH.

Ingress/egress policies. Recall that the ingress and egress policies determine which data flows are allowed and reflect the policies of users, data sources, and the provider. In our system, the network connection between the client and the front-end is both an ingress and an egress conduit (for search queries and search results, respectively). The document files in the corpus and the key-value tuples that contain a user's personal information are ingress conduits. Policies are associated with all ingress and egress conduits as described below. The primary difficulty here is to determine appropriate policies, a task that is required in any compliant system. Specifying the policies in THOTH's policy language is straightforward.

Account manager flow. When Alice creates an account, credentials are exchanged for subsequent mutual authentication, and stored in the key-value store, along with any personal profile information Alice provides.

Alice can choose policies for all the ingress conduits she controls, typically from a set of policy templates written by the provider's compliance team. She can choose policies for her profile entries, controlling how the personal information she provides can be used, as well as any information the system collects (e.g., Alice's search queries, her clicks, or the IP addresses from which she contacts the search engine). For instance, she can choose whether her query or click history, or any other part of her profile information may be used to personalize search results and target ads.

If Alice contributes content, she can choose a policy for each item. The declassification clause on each policy implicitly controls who can subsequently change the policy; normally, Alice would choose a policy that allows only her to make such a change. Alice may also edit her friend list stored in the key-value store, which may be referenced by her policies.

The account manager process must be trusted by Alice to install policies according to her wishes and to use her authentication credentials appropriately. Such a trusted component is necessary, but could be shifted to the client's computer by allowing a remote process to interact directly with **T**HOTH. No other user process in the system is trusted by Alice.

The provider also associates its policies with relevant ingress and egress conduits. For instance, the provider requires censorship and MAL policies on all indexed documents, and adds these to the policy templates available to users.

Indexing flow. Periodically, the indexer is invoked to regenerate the index partitions. A correct indexer processes only documents with the declassification clause ONLY_CND_IDS (or ONLY_CND_IDS+), which in turn is transferred to the index files. Note that the index may contain arbitrary data and can be read by any `CONFINED` process; however, an eventual declassification to an `UNCONFINED` process is only possible by writing a list of conduit ids.

Profile aggregation flow. A profile aggregation task periodically executes in the background, to scan the suffix of a user's query and click history and to update the user's profile vector. A correct aggregator only analyzes user history data that has the declassification clause ONLY_CND_IDS (or ONLY_CND_IDS+), which is transferred to the profile vectors.

Search flow. Finally, we describe the sequence of steps when Alice performs a search query. The search front-end authenticates itself to Alice using the credentials stored in the key-value store. A successful authentication assures Alice that *(i)* she is talking to the front-end, and *(ii)* the front-end process is tainted with the policy of Alice's credentials (only Alice can read, else declassify into a list of conduit ids) before Alice sends her search query. Next, Alice authenticates herself to the THOTH reference monitor via the search front-end, which proves to THOTH that the front-end process speaks for Alice.

The front-end now may rewrite the query based on the information in Alice's profile, before it sends Alice's query to one or more search process(es) and adds it to her search history. The search results are declassified as a list of conduit ids, and therefore do not add new taint to the front-end. While producing the HTML results page, the front-end reads a snippet from each result document using Alice's credentials. Each document has a censorship policy, which checks that the document's conduit ID is not blacklisted in the client's region. These policies differ in the conduit IDs and so, in principle, the taint set on the front-end could become very large. To prevent this, we use partial evaluation (Subsection 3.1.5): *Before* a document's policy is added to the front-end's taint, we check that the document is not blacklisted. This way, the front-end's taint increases by a *single predicate* (which verifies Alice's IP address) when it reads the first document and does not increase when it reads subsequent documents.

Finally, the front-end sends the results page to the client. For this, it must satisfy the egress conduit policy, which verifies Alice's identity and her IP address.

Result caching. High-performance retrieval systems cache search results and content snippets for reuse in similar queries. Although we have not implemented such caching, it can be supported by THOTH. Intermediate results can be cached at various points in the data flow, usually before their policies have been specialized (through partial evaluation) for a particular client or jurisdiction.

Summary. Assuming that the account manager correctly installs ingress and egress policies, THOTH ensures that Alice's documents, history and profile are used according to her wishes and that the provider's censorship and MAL policies are enforced, despite any bugs in the indexer, the front-end or the profile aggregator. THOTH's use in a data retrieval system highlights two different ways of preventing process overtainting. The front-end process is *user-specific*—it acts on behalf of one client. Consequently, the front-end must be re-exec'ed at the end of a user session session to discard its taint. In contrast, the indexer is an *aggregator* process that is designed to combine documents with conflicting policies into a single index. To make its output (the index) usable downstream, the provider installs a typed declassification clause (ONLY_CND_IDS or ONLY_CND_IDS+) on all documents. Due to the declassification clause, there is no need to re-exec the search process.

Data Retrieval with THOTH: Implementation

In this section, we describe our experience implementing the data retrieval system. We provide commentary on the code changes we introduced to the baseline system in order to allow flows while running within **THOTH**. Since **THOTH**'s policy enforcement is independent of application code, these changes are purely to make the data flows comply with the data use policies in place or to reduce the overhead of policy enforcement.

With **THOTH**, the front-end, search, indexing, and aggregation tasks execute as `CONFINED` processes, and the account manager executes as an `UNCONFINED` process. Relative to the baseline system, we made minimal modifications, mostly to set an appropriate policy on output conduits. As we have noted earlier, the data retrieval system is built around Apache Lucene. The modifications to Apache Lucene amounted to less than 20 lines of Java code and 30 lines of C code in a JNI library. These modifications set policies on internal conduits and, like the rest of Lucene, are not trusted. Finding the appropriate points to modify was relatively easy because Lucene's codebase has separate functions through which all I/O is channelled. (For applications without this modularity, a dynamically-linked library can be used that overrides libc's I/O functions and adds appropriate policies.)

Indexer Process. We have introduced three changes to the stock Apache Lucene indexer.

- **Transition the indexer's state.** The indexer must run as a `CONFINED` process since it cannot satisfy the read rules of the searchable content's policies such as the private and friends-only policies.
- **Set the indexer's taint.** As the indexer consumes searchable content, their policies are checked for restrictiveness against the indexer's current taint. If the current taint is less restrictive, the policies get added to the taint (or dropped otherwise). The taint size increases as the indexer consumes more content, causing the restrictiveness check to become more expensive. To avoid this problem, the indexer sets its taint to the index policy (see Subsection 3.2.2). With this policy as taint, the taint size does not increase as it is more restrictive than the policies of all searchable content.
- **Set policies appropriately on the index files.** The indexer attaches the index policy to newly created index files. (Otherwise, writes are denied if the created files' polices are

not at least as restrictive as the indexer's taint). Lucene's codebase is modularized, and therefore, our changes to set policies on the output index files were limited to a few places.

Search Process. We hightlight two changes that are required to make the I/O of the search process policy compliant.

- **Transition the search process's state.** The search process must run in a `CONFINED` state as it cannot satisfy the read rule of the index policy. Once it consumes the index, its taint gets the index policy.
- **Set policies appropriately on search output.** The search process's taint limits declassification to ONLY_CND_ID (or to ONLY_CND_ID+). Thus, the policy attached to the results conduit (to which the search process writes the conduits ids) must be at least as restrictive as all the policies of the conduits whose ids are being written. For this reason, the search process creates and sets a policy on the results conduit based on the conduits ids included in the result. We extended THOTH's API calls to query conduits' policies in a batch to do this efficiently.

Front-end Processes. Front-end processes are not part of Apache Lucene's codebase. Our implementation mimics a simple web server that *(i)* accepts an external user's connection, *(ii)* submits user's queries to the search process over a pipe, and *(iii)* creates a search results' snippet based on the search engine output and sends the snippet to the external user. We next describe core functionalities that are necessary for the front-end to function with THOTH.

- **Cache system's meta-data for policy evaluation.** The front-end process may need to perform some actions in order to satisfy the policies of conduits it needs to access. For instance, Alice's front-end needs to cache Alice's entry in Bob's friends list before accessing Bob's friends-only files (via the CACHE(...) API call). This is necessary so that THOTH's policy evaluation allows the access. Similarly, a front-end needs to cache parts of a region's blacklist file when accessing content subject to region-based censorship. THOTH provides a library to facilitate these actions, requiring few application code changes.
- **Exec after session termination.** Unlike in the baseline, the front-end process must be restarted after each user session under THOTH, to shed its taint. We implement this taint shedding by exec-ing the process when a new user session starts.

3.5 THOTH Evaluation

In this section, we present results of an experimental evaluation of our THOTH prototype.

Evaluation setup. All experiments were performed on Dell R410 servers, each with 2x Intel Xeon X5650 2.66 GHz 6 hyperthreaded core CPUs, 48GB main memory, running OpenSuse Linux 12.1 (kernel version 3.13.1, x86-64). The servers are connected to Cisco Nexus 7018 switches with 1Gbit Ethernet links. Each server has a 1TB Seagate ST31000424SS disk formatted under ext4, which contains the OS installation and a 258GB static snapshot of English language Wikipedia articles from 2008 [91].

We allocate a 2GB memory segment on `/dev/shm` to simulate NVRAM used by THOTH to store its metadata and transaction log. NVRAM is readily available and commonly used to store frequently updated, fixed-sized persistent data structures like transaction logs.

In the following experiments, we compare a system where each OS kernel is configured with the THOTH LSM kernel module and reference monitor against an otherwise identical baseline system with unmodified Linux 3.13.1 kernels. This baseline configuration offers no policy protection.

3.5.1 THOTH-based data retrieval system

First, we study the total THOTH overheads in the prototype retrieval system described in Section 3.4.

Indexing

First, we measure the overhead of the search engine's index computation. We run the Lucene indexer over *(a)* the entire 258GB snapshot of the English Wikipedia, and *(b)* a 5GB part of the snapshot. The sizes of the resulting indices are 54GB and 959MB, respectively. Table 3.3 shows the average indexing time and standard deviation across 3 runs. In both cases, THOTH's runtime overhead is below 1%.

Even in a sharded configuration, Lucene relies on a sequential indexer, which can become a bottleneck when a corpus is large and dynamic. Larger search engines may rely on parallel map/reduce jobs to produce their index. As a proof of concept, we built a Hadoop-based indexer using THOTH, although we don't use it in the following evaluation because it does

	Dataset 258GB		Dataset 5GB	
	Avg. (mins)	σ	Avg. (mins)	σ
Linux	1956.1	30	27.8	0.06
THOTH	1968.6	24	28.0	0.11
Overhead	0.65%		0.7%	

TABLE 3.3: Indexing runtime overhead with THOTH.

not support all the features of the Lucene indexer. All mappers and reducers run as confined tasks, and receive the same taint as the original, sequential indexer.

Search throughput

Next, we measure the overhead of THOTH on search throughput. To ensure load balance, we partitioned the index into two shards of 22GB and 33GB, chosen to achieve approximately equal query throughput. We use two configurations:

- **2SERVERS**: 2 server machines execute a Lucene instance with different index shards.
- **4SERVERS**: Here, we use two replicated Lucene instances in each shard to scale the throughput. The front-end forwards each search request to one of the two Lucene instances in each shard and merges the results.

We drive the experiment with the following workload. We simulate a population of 40,000 users, where each user is assigned a friend list consisting of 12 randomly chosen other users, subject to the constraint that the friendship relationship is symmetric. Each item in the corpus is assigned either a private, public, or friends-only policy in the proportion 30/50/20%, respectively. A total of 1.0% of the entire dataset is censored in some region. All simulated clients are in a region that blacklists 2250 random items of the dataset.

We use query strings based on the popularity of Wikipedia page accesses during one hour on April 1, 2012 [90]. Specifically, we search for the titles of the top 20K visited articles and assign each of the queries randomly to one of the users. 24 simulated active users connect to each server machine, maintain their sessions throughout the experiment, and issue 48 (**2SERVERS**) and 96 (**4SERVERS**) queries concurrently to saturate the system. In addition, a simulated "employee" sporadically issues a read access to protected user files for a total of 200 MAL accesses.

During each query, the front-end looks up the user profile and updates the user's search history in the key-value store. To maximize the performance of the baseline and to fully expose

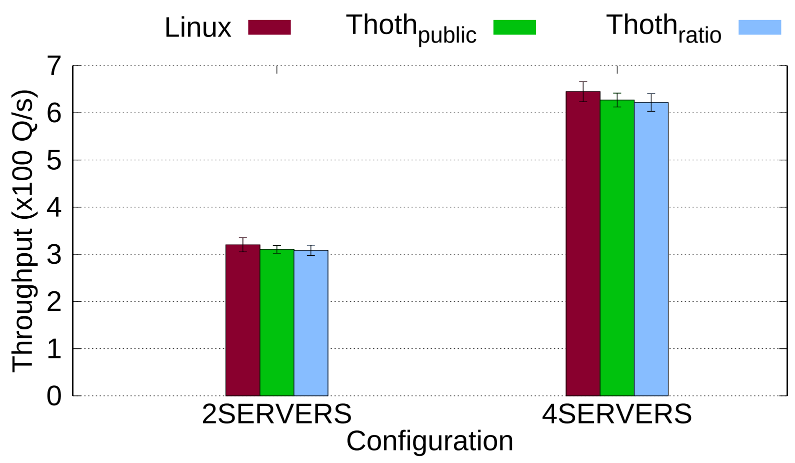


FIGURE 3.3: Average search throughput in queries per second of 48 and 96 concurrent users (**2SERVERS** and **4SERVERS**, respectively). Users maintain their sessions for the duration of the experiment. Error bars show standard deviation.

THOTH's overheads, the index shard and parts of the corpus relevant to our query stream are pre-loaded into the servers' main memory caches, resulting in a CPU-bound workload.

Figure 3.3 shows the average throughput over 10 runs of 20,000 queries each, for the baseline (Linux) and THOTH under **2SERVERS** and **4SERVERS**. The error bars indicate the standard deviation over the 10 runs. We used two THOTH configurations, **THOTH$_{\textbf{public}}$** and **THOTH$_{\textbf{ratio}}$**.

- **THOTH$_{\textbf{public}}$**: The policies permit all accesses. This configuration helps to isolate the overhead of THOTH's I/O interposition and reference monitor invocation for the null policy check.
- **THOTH$_{\textbf{ratio}}$**: The policies of input files are private to a user, public, or accessible to friends-only following the ratio 30:50:20. Moreover, all files allow employee access under MAL, enforce region-based censorship, and have the declassification condition with ONLY_CONDUIT_IDS+.

The query throughput scales approximately linearly moving from **2SERVERS** (320 Q/s) to **4SERVERS** (644 Q/s), as expected. THOTH with all policies enforced (**THOTH$_{\textbf{ratio}}$**) has an overhead of 3.63% (308 Q/s) in **2SERVERS** and 3.55% in **4SERVERS** (621 Q/s). We note that the throughput achieved with **THOTH$_{\textbf{public}}$** (310 Q/s and 627 Q/s, respectively) is only slightly higher than **THOTH$_{\textbf{ratio}}$**'s. This indicates that THOTH's overhead is dominated by costs like I/O interception, THOTH API calls, and metadata operations, which are unrelated to policy

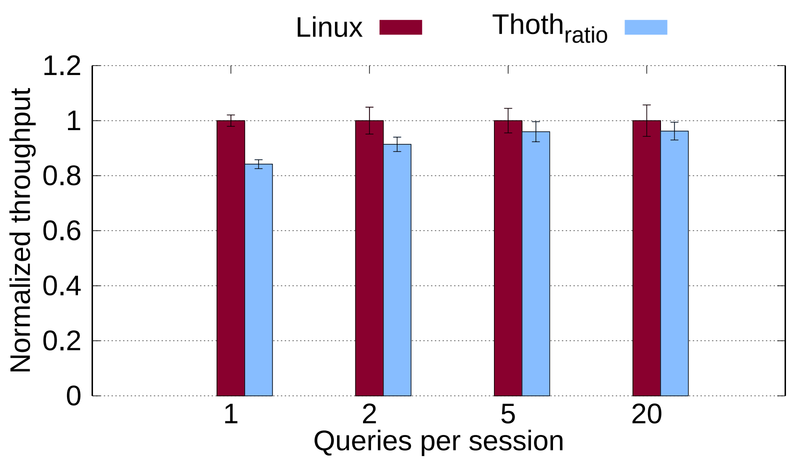


FIGURE 3.4: Average search throughput of 48 concurrent users under **2SERVERS**, normalized to the baseline Linux . Users maintain their sessions for lengths 1, 2, 5, and 20 queries. Error bars show standard deviation.

complexity. Therefore, changing the policy configuration to different ratios of public, private, or friends-only will have little effect on **THOTH**'s overhead presented in this experiment.

With **THOTH**, the front-end is re-exec'ed at the end of every user session to shed the front-end's taint. The relative overhead of doing so reduces with session length. Figure 3.4 shows the average throughput normalized to the Linux baseline for session lengths of 1, 2, 5 and 20 queries in **2SERVERS**. With **THOTH**, the front-end process cannot be used across sessions, because it is tainted with a user's policies. Therefore, front-end processes must be re-exec'ed before accepting new user sessions. Due to the per-session front-end exec, **THOTH**'s overhead is higher for small sessions (15.8% for a single query per session); however, the overhead diminishes quickly to 8.6% for 2 queries per session, 4.0% for 5 queries per session, and 3.8% for 20 queries per session. Overall, the throughput of 5 or more queries per session is within a standard deviation from the achieved throughput when users maintain their sessions during the whole experiment. This holds in all configurations, including **4SERVERS**.

Search latency

Next, we measure the overhead on query latency. Table 3.4 shows the average query latency across 5 runs of 10,000 queries in **2SERVERS**. Here, a single user connects to a front-end, submits a search query, and disconnects after receiving the search results snippet. We repeated the experiment under **4SERVERS**, and the results are similar. In all cases, **THOTH** adds less than 6.7ms to the baseline latency.

	Avg. (ms)	σ	Overhead
`Linux`	47.09	0.43	
THOTH$_{\textbf{public}}$	51.60	0.29	9.6%
THOTH$_{\textbf{ratio}}$	53.78	0.20	14.2%

TABLE 3.4: Average query search latency in milliseconds for a single user under **2SERVERS**. The user re-establishes a new session with each search query.

Policy enforcement overhead breakdown

There are multiple sources contributing to **T**HOTH's runtime overhead.

- *I/O interception:* The LSM intercepts processes' I/O. This overhead is mostly re-directing control flow in the kernel to the LSM and allocating memory to hold arguments (such as process identifiers, conduits' pathnames, and content if needed).
- *IPC to the reference monitor:* The LSM invokes the userspace reference monitor process when processes perform I/O or issue system calls that may require action by the reference monitor, resulting in IPC and context switches.
- *Policy lookup:* The reference monitor looks up the policy store to find which (if any) policy is attached to a given conduit.
- *Policy evaluation:* The reference monitor performs policy evaluation to determine if access is policy-compliant (or to check for restrictiveness when tracking taint).
- *Session isolation:* A front-end process must re-exec before starting a new user session.

For instance, out of **THOTH$_{\textbf{ratio}}$**'s overhead of 3.8% on search throughput under 20 queries per session: I/O interception is 0.4%; IPC to reference monitor is 1.6%; policy lookup is 1.0%; policy evaluation is 0.5%; and session isolation is 0.3%.

While I/O interception and IPC to the reference monitor contribute most to the overhead, the overhead due to policy lookup and evaluation is also not negligible. Moreover, the session isolation cost increases as user sessions become shorter. Hence, to drastically reduce the runtime overhead, **T**HOTH must optimize performance across these different dimensions.

As a proof of concept, we implemented a rudimentary reference monitor in the kernel, which does not support session management and policy interpretation (which require libraries that are unavailable in the Linux kernel). This reduced in-kernel monitor suffices to execute **THOTH$_{\textbf{public}}$** when users maintain their sessions during the whole experiment. Moving the reference monitor to the kernel reduced the overhead of **THOTH$_{\textbf{public}}$** from 3% to under 1%, due to eliminating the cost of IPC between the LSM and the reference monitor. While this suggests that **T**HOTH's overheads can be further reduced using careful engineering, it also indicates that **T**HOTH's runtime overhead cannot be close to zero. In the next chapter, we show how **S**HAI is able to effectively eliminate many sources of **T**HOTH's overhead, enforcing data use policies with near-zero runtime overhead on the critical path.

3.5.2 Microbenchmarks

Next, we perform a set of microbenchmarks to isolate **T**HOTH's overheads on different policies. We measure the latency of opening, reading sequentially, and closing 10,000 files in the baseline and with **T**HOTH under different policies associated with the files. The files were previously written to disk sequentially to ensure fast sequential read performance for the baseline, fully exposing the overheads.

In the **T**HOTH experiments, accesses are performed by an `UNCONFINED` task to force an immediate policy evaluation. The following policies are used:

- **THOTH$_{\textbf{public}}$**: files can be read by anyone.
- **THOTH$_{\textbf{private}}$**: access is restricted to a specific user.
- **THOTH$_{\textbf{ACL}}$**: access to friends only (all users have the same friend list).
- **THOTH$_{\textbf{ACL+}}$**: access to friends only (each user has a different friend list).
- **THOTH$_{\textbf{FoF}}$**: access to friends of friends (each user has a different friend list). All friend lists used in the microbenchmark have 100 entries.
- **THOTH$_{\textbf{MAL}}$**: each file has a MAL policy, where each read requires an entry in a log with an append-only integrity policy.

Figure 3.5 shows the average time for reading a file of sizes 4KB and 512KB, normalized to the baseline Linux latency (0.145ms and 3.6ms, respectively); the error bars indicate the standard deviation among the 10,000 file reads. We see that **T**HOTH's overheads increase with the complexity of the policy, in the order listed above. For the 4KB files, the overheads range from 10.6% for **THOTH$_{\textbf{public}}$** and **THOTH$_{\textbf{private}}$** to 152.7% for **THOTH$_{\textbf{MAL}}$**. The same trend holds for larger files, but the overhead range diminishes to 0.6%–23% for 96KB files (not shown in the figure) and 0.34%–3.3% for 512KB files.

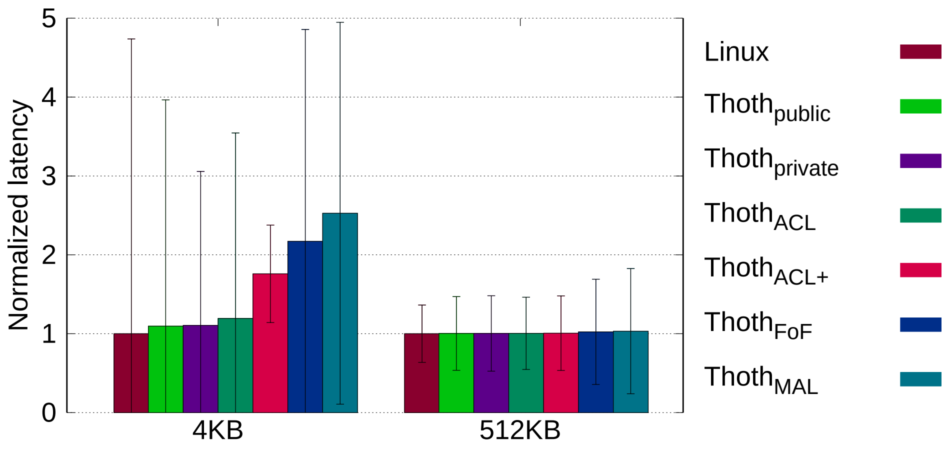


FIGURE 3.5: Read latency, normalized to Linux's.

We also experimented with friend list sizes of 12 and 50 entries for the configurations **THOTH$_{ACL}$**, **THOTH$_{ACL+}$** and **THOTH$_{FoF}$** under files sizes 4KB and 512KB; the resulting latency was within 2.4% of the corresponding 100-entry friend list latency in all cases. This is consistent with the known complexity of the friend lookup, which is logarithmic in the list size.

We also looked at the breakdown of THOTH latency overheads. With **THOTH$_{ACL}$** and 4KB files, THOTH's overhead for file read is on average 28µs, which are spent intercepting the system call and maintaining the session state. Interpreting the policy and checking the friend lists takes 6µs, but this time is completely overlapped with the disk read.

Write transaction latency. We performed similar microbenchmarks for write transactions. In general, THOTH's write transactions have low overhead since its transaction log is stored in (simulated) NVRAM. As in the case of read latency, the overhead depends on the granularity of writes and the complexity of the policy being enforced. Under the index policy, the overhead ranges from 0.25% for creating large files to 2.53x in the case of small files. The baseline Linux is very fast at creating small files that are written to disk asynchronously, while THOTH has to synchronously update its policy store when a new file is created. The overhead is 5.8x and 8.6x in the case of a write of 10 conduit ids to a file under the ONLY_CND_IDS and ONLY_CND_IDS+ policies, respectively. This high overhead is due to checking that each conduit id being written exists (and is written into a file with a stricter policy in the case of ONLY_CND_IDS+). However, this overhead amounts to only a small percentage of the overall search query processing, as is evident from Table 3.4.

3.5.3 Fault-injection tests

To double-check THOTH's ability to stop unwanted data leaks, we injected several types of faults in different stages of the search pipeline.

Faulty Lucene indexer. We reproduced a known Lucene bug [5] that associates documents with wrong attributes during index creation. This bug is security-relevant because, in the absence of another mechanism, attributes can be used for labeling data with their owners. In our experiment THOTH successfully stopped the flow in all cases where the search results contained a conduit whose policy disallowed access to the client.

We also intentionally misconfigured the indexer to index the users' query and click histories, which should not show up in search results. THOTH prevented the indexer from writing the index after it had read either the query or the click history.

Faulty Lucene search. We reproduced a number of known Lucene bugs that produce incorrect search results. Such bugs may produce Alice's private documents in Bob's search. The bugs include incorrect parsing of special characters [9], incorrect tokenization [4], confusing uppercase and lowercase letters [10], using an incorrect logic for query expansion [6, 7], applying incorrect keyword filters [8], and premature search termination [11]. We confirmed that all policy violations resulting from these bugs were blocked by THOTH.

To check the declassification condition ONLY_CND_IDS+, we modified the search process to (incorrectly) output text from the index in place of conduit ids. THOTH prevented the search process from producing such output.

Faulty front-end. We issued accesses to a private file protected by the MAL policy without adding appropriate log entries. THOTH prevented the front-end process from extricating data to the caller. We performed similar tests for the region-based censorship policy with similar results.

3.6 THOTH Conclusion

Efficient policy compliance in data retrieval systems is a challenging problem. THOTH is a kernel-level policy compliance layer to address this problem. The provider has the option to associate a declarative policy with any data conduit, in particular, with any data source. The policy specifies all the confidentiality and integrity requirements in effect on data in the conduit. The policy language is rich and can express data owner's privacy preferences, the provider's own data-use policy, and legal requirements. THOTH enforces these policies by tracking and controlling data flows across tasks through kernel I/O interception. It prevents data leaks and corruption due to bugs and misconfigurations in application components (including misconfigurations in policies on internal conduits), as well as actions by unprivileged operators.

THOTH's technical contributions include a declarative policy language that specifies both access (read/write) policies and how those access policies may change. The latter can be used to represent declassification policies. Additionally, the language supports content-dependent policies. THOTH uses policy sets as taint, which eliminates the need to trust application processes with access checks at the system boundary and with declassification.

Our Linux-based prototype shows that THOTH can be deployed with low overhead, and is suitable for data retrieval systems that need to sustain throughput up to a few hundred search requests/second/machine. This demonstrates the usefulness and viability of coarse-grained taint tracking as a basis for policy enforcement.

Nonetheless, we see THOTH as the first milestone in the design space of comprehensive and practical policy compliance systems. There is still work to be done. In particular, THOTH is not suitable for data retrieval systems that need to sustain more than a few hundreds of search requests/second/machine. This limitation is fundamental to THOTH's design: THOTH relies on runtime monitoring (I/O interception and taint tracking), and the monitoring overhead increases as applications perform more I/O. In the next chapter, we investigate a policy compliance design that is able to achieve a runtime overhead close to zero when serving users' requests under workloads with thousands or even tens of thousands of search requests/second/machine.

4 SHAI: Policy Compliance via Offline Analysis and Light-Weight Runtime Monitoring

In the previous chapter, we discussed THOTH, which enforces policies via runtime monitoring at the expense of incurring runtime overhead. In this chapter, we present another policy compliance system, SHAI, whose goal is to reduce the policy enforcement runtime overhead on the critical path (e.g., when serving users' requests) to near-zero. THOTH's runtime overhead is due to multiple factors: I/O interception, IPC to the reference monitor, policy lookup and evaluation, and session isolation. In order to effectively reduce the runtime overhead, the compliance system must mitigate all these factors. We start with a discussion of techniques to mitigate the aforementioned overhead factors. Afterwards, we present how SHAI adopts such techniques.

One of the main overhead factors in THOTH is I/O interception. THOTH intercepts tasks' reads and writes for taint tracking and flow control. Such I/O interception is necessary since a task in THOTH may attempt to read from or to write to *any* conduit. In other words, THOTH relies on I/O interception since it does not make any assumptions about tasks' reads or writes. Assume, for the time being, that tasks' set of accesses can be perfectly predicted. In such case, tasks' taints can be computed ahead of time based on the conduits which the tasks are predicted to read. Moreover, a task's predicted writes could be checked against its taint ahead of time as well. Here, policy compliance can be determined completely offline.

In this proposed model where policy compliance is determined offline based on the predicated set of accesses, policies can be enforced at runtime by limiting tasks' reads and writes to the set of predicted, compliant accesses. This can be done, for instance, by running tasks in sandboxes.

This proposed model of policy enforcement has a runtime overhead profile different from that of **T**HOTH's. Recall that the main factors of **T**HOTH's overhead are I/O interception, IPC to the reference monitor, policy lookup and evaluation, and session isolation. While **T**HOTH relies on I/O interception for taint tracking and flow control, this proposed model avoids I/O interception and, in turn, does away with the IPC to the reference monitor. It also performs policy lookup and evaluation offline, off the hot path. This is a good start. We are now left with one remaining runtime overhead factor, namely session isolation. **T**HOTH uses OS processes to isolate user sessions. While session isolation for tasks that can potentially serve multiple users is fundamental to coarse-grained information flow control, this overhead can be reduced by relying on efficient in-process isolation primitives such as *lw*Cs [52].

The proposed model offers high-level insights that can mitigate **T**HOTH's runtime overhead. In reality, however, the premise that the tasks' set of accesses and their taints can be perfectly predicted ahead of time does not always hold. For instance, the taint of a front-end process in the search pipeline presented in Subsection 3.4 cannot be perfectly predicted ahead of time. This taint depends on the identity of the connected user and the user's geographic location, and both are available only at runtime. Moreover, changes in policies or in content state (e.g., friends lists) may cause mispredictions. Therefore, relying on these insights comes with the challenge of operating under incomplete information and mispredictions. Next, we present **S**HAI which relies on these insights while addressing the accompanying challenges in order to reduce the policy enforcement runtime overhead to near-zero.

SHAI is a novel system for policy compliance that can enforce fine-grained, data- and user-specific declarative flow policies with near-zero runtime overhead in the common case. **S**HAI combines *offline analysis* and *light-weight runtime monitoring* using an operating system's capability sandbox. The idea behind **S**HAI is straightforward: It pushes as much work as possible to the offline analysis to minimize and streamline the remaining, required runtime monitoring. The design of **S**HAI is based on the following ideas:

Use of offline analysis. Many aspects of a data retrieval system's runtime behavior can be predicted offline based on testing, monitoring the production pipeline or manual analysis. These aspects include the normal flow of information among the system's components (tasks), the set of policies currently in effect, the set of users, and the geographic region(s) from which a user typically connects. Based on this information, an offline analysis (OA) predicts the taint each of the system's tasks will acquire at runtime, subject to assumptions about the values of runtime variables. Finally, the OA compiles, for each task and each predicted runtime value, the predicted taint into a set of capabilities for all compliant I/O accesses.

Session-level binding of runtime information. Many runtime variables that are unknown during an offline analysis become known at the start of a user session. These variables include the identity of the user, the geographic region from which the user connects, and the wall-clock time. Based on the actual values of these variables, SHAI assigns the appropriate capability set provided by the OA for each task involved. If the value of a runtime variable is not among those predicted during the OA, SHAI registers the value as one that should be considered during the next OA.

OS sandbox to allow compliant I/O without runtime intervention. In SHAI, each of the system's tasks is encapsulated in an OS sandbox subject to capability-based I/O access control. When a user session starts, SHAI grants each task the capability set predicted by the OA and selected based on available runtime information. As a result, the system can perform compliant accesses without runtime intervention. Because the capability checks are light-weight and performed by the OS kernel, their overhead is very low. In the common case where the runtime values are among those predicted by the OA, the cost of enforcing compliance is near-zero.

Runtime reference monitor only as a fall-back. If a task performs an I/O access for which it does not have a valid capability, control is transferred to the SHAI reference monitor (RM), which performs a runtime policy check. The cause of this event may be a non-compliant access, an imprecise OA, or a change in the system state since the OA was performed (e.g., a change in policy, or an access to content that was created after the OA). If the access is non-compliant, the RM denies the offending access. If the access turns out to be compliant, the RM allows the access and patches the task's capability set to reflect the latest set of compliant accesses.

Use of efficient OS isolation primitives. As mentioned earlier, SHAI uses light-weight contexts (*lwC*s) [52], an efficient OS isolation primitive, to isolate multiple user sessions within the same process, and to isolate SHAI's reference monitor.

The rest of this chapter is organized as follows. We provide an overview of SHAI and its components in Section 4.1 and a detailed description of SHAI's design in Section 4.2. The SHAI prototype implementation on FreeBSD and the application to a data retrieval system based on Apache Lucene are described in Section 4.3 and Section 4.4, respectively. We present the results of an experimental evaluation in Section 4.5. Finally, we conclude in Section 4.6.

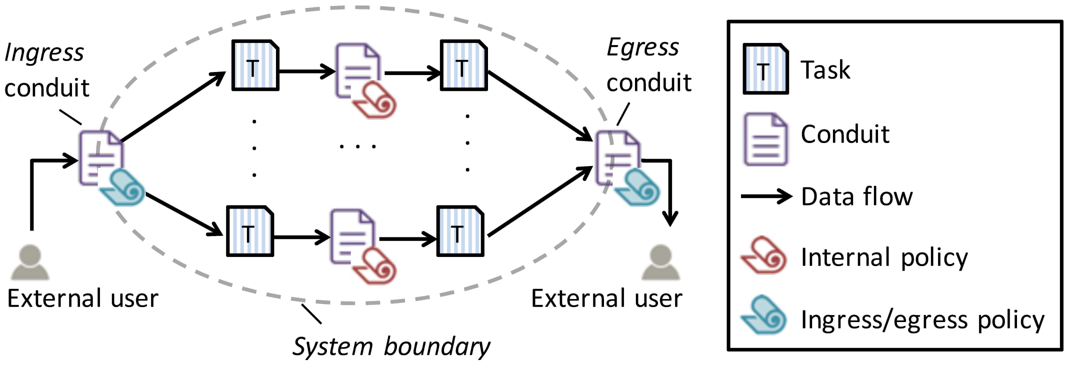


FIGURE 4.1: SHAI data flow model.

4.1 SHAI Overview

SHAI and THOTH share the same goal (help data retrieval system providers enforce confidentiality and integrity policies on the data they collect and serve) and many design principles (declarative policies directly attached to data, policy enforcement separate from application code). However, SHAI uses different architecture and policy enforcement mechanisms with the goal of achieving *near-zero runtime overhead* for policy enforcement in the common case. We next describe SHAI's design, overall architecture, and threat model.

4.1.1 Data flow model and policy language

SHAI and THOTH share the same data flow model and policy language. Both have been covered in detail in Subsection 3.1.1 and Subsection 3.1.2, respectively. We summarize them briefly next, but the readers may skip over this part if such details are fresh in their mind.

Figure 4.1 shows SHAI's data flow model. SHAI enforces data policies in systems that are structured as pipelines of *tasks*. Each task in the pipeline consumes some data, processes it, and produces more data, which is then consumed by the next task in the pipeline. Data enters the pipeline, travels from one task to the next, and eventually leaves the pipeline in *conduits*, which is a generic abstraction for any container of data (e.g., files, named pipes). Every conduit has a unique identifier (e.g., full path name for a file). We distinguish three kinds of conduits: *ingress conduits* that feed outside data to the initial tasks of the pipeline, *internal conduits* that are used to pass data between tasks of the pipeline, and *egress conduits* that are used to transmit final outputs of the pipeline to external applications or externally connected users.

An administrator may associate a policy with any conduit. This policy is a *one-point description* of all the confidentiality and integrity requirements of the data in the conduit. Policies are specified in the same declarative policy language used in THOTH. In this language, a conduit's policy has three rules: 1) A **read** rule specifies who can read the conduit's data directly; 2) A **declassify** rule specifies what **read** rules should apply to conduits downstream in the pipeline, thus controlling who can read derived data. The **declassify** rule specifies a set of tests (called *declassification conditions*) on the global state and data in any conduit downstream, and how the **read** rule can be relaxed when each of those tests is satisfied; 3) An **update** rule specifies what type of content can be written to the conduit and by whom.

Policies on inbound and outbound conduits that cross the system boundary represent the ingress and egress policies, respectively. These ingress and egress policies collectively control how data can be used within the system, and how the data can be disseminated (and fed) from (and into) the system.

4.1.2 Runtime overhead sources in THOTH

The starting point for our design is THOTH, which can already enforce data-specific policies efficiently in low throughput systems. However, THOTH's overheads can be substantial in high throughput systems. In the following, we review briefly how THOTH works, what the dominant sources of overhead in THOTH are, and what SHAI does differently to mitigate these overheads.

THOTH performs *coarse-grained runtime flow tracking* to enforce policies. THOTH maps tasks to OS processes and implements a reference monitor (RM) that intercepts every conduit I/O in the kernel. The RM maintains a *taint* for every task (process) in the pipeline. This taint is actually a policy that is always at least as restrictive as the policies of all conduits that the task has read in the past.

When a task opens a conduit for reading, the RM intercepts to check whether the taint on the task is already more restrictive than the policy of the conduit. If so, it does nothing further. If not, it intersects the current taint of the task with the policy of the conduit. When a task opens a conduit for writing, the RM intercepts to first check the **update** rule of the conduit. Next, it checks the declassification conditions in the taint of the task, which may relax the taint, and then checks whether the (possibly relaxed) taint is at least as permissive as the policy of the conduit being written. These checks on conduit opens ensure that, modulo declassification, the policies of conduits downstream of a conduit f are always more restrictive than f's policy. As a result, the restrictions of f's policy cannot be "lost" on data derived downstream.

The RM *enforces* policies on egress conduits connected to end-users by direct checks. For example, if an egress conduit's **read** policy says that only Alice can read, then the RM ensures that the egress conduit is actually connected to Alice by verifying the public key that authenticates the connection.

Despite its efficiency compared to older solutions, **T**HOTH still has overhead with respect to the system's throughput, especially on moderate or high throughput systems. As a case in point, **T**HOTH has a relative overhead of 3.63% on the throughput of a simple data indexing and search pipeline, even at a modest throughput of only ~300 queries/second/machine. As the throughput increases, this relative overhead increases significantly, reaching over 23% at ~3,000 queries/second/machine on a port of **T**HOTH to the experimental setup used for **S**HAI (see Section 4.5). In Subsection 3.5.1, we have provided a detailed breakdown of **T**HOTH's runtime overhead. Briefly, this runtime overhead is due to the following sources.

1. I/O interception by the RM to check taints and declassification conditions is expensive. In **T**HOTH, every interception involves a context switch to a dedicated process that hosts the RM.
2. Looking up conduits' policies, maintaining tasks' taint, and performing policy evaluation, *all* at runtime.
3. Once a user-facing task has served private data to a user, that task cannot serve a different user without shedding its previous taint. To shed that taint cleanly, the task must be reset to a clean state. The usual way of doing this, also used in **T**HOTH, is to re-exec the process hosting the task. Re-execing is expensive. Since taint must be shed only once per user session, the amortized cost of re-exec'ing reduces with increase in session length, but it is still significant even for moderate session lengths (5-20 queries per session) in **T**HOTH.

4.1.3 Key ideas

SHAI is a *re-design* of **T**HOTH with two key ideas to mitigate most of **T**HOTH's overhead. First, **S**HAI adds to **T**HOTH a new offline phase that does most of the work of the RM ahead-of-time, thus significantly reducing the need to intercept I/O and pushes most policy lookups and evaluation to the background. Second, **S**HAI uses a different implementation of tasks that allows for much faster state reset. End-to-end, the offline phase reduces the overhead on each user request to near-zero in the common case, while the change to the implementation of tasks significantly reduces the overhead on user session establishment.

Eliminating I/O interceptions and policy operations

SHAI eliminates the need for runtime interception and policy lookup and evaluation of most conduit accesses using a periodic, ahead-of-time, offline analysis (OA). During the OA, **S**HAI makes (and caches) policy checks on the reads and writes that the system is likely to make in later executions of the pipeline. For this, the OA takes as input a list of tasks in the pipeline, what conduits each task is likely to read and write during the pipeline's execution, an estimate of the task's anticipated taint at runtime and the policies of all conduits. With the exception of the policies (which are set by the provider and we expect to correctly capture the integrity and confidentiality requirement of data), these inputs are *not trusted* for policy enforcement; getting some of them wrong only results in a proportionally higher overhead at runtime. All inputs can be easily determined by running the pipeline in a test environment, by monitoring the production system, or by a simple manual analysis.

The OA simulates the checks that **T**HOTH's RM would make for each conduit access specified in the inputs, but without actually running the pipeline. Later, at runtime, each task runs in an *OS sandbox*, which allows the *certified accesses* that were already checked by the OA without faulting into the RM. These accesses run at native speed since the checks of the OS sandbox are streamlined in the underlying kernel. In the (rare) case that an access not foreseen by the OA occurs, the OS sandbox faults into the RM, which makes the same policy checks that **T**HOTH would make.

Our current prototype uses FreeBSD's capability system (Capsicum) [89] (with small modifications) for the OS sandbox. Capability checks in Capsicum are highly optimized and incur nearly zero overhead. This, coupled with the OA, pushes most of policy operations (lookup and evaluation) to the background and reduces the overhead of I/O interception to nearly zero in the common case.

While this idea is conceptually simple, it has several nuanced details that we explain in Section 4.2. First, a task's ability to make certain accesses may depend on parameters whose exact values will be known only at runtime, e.g., which user has authenticated remotely, which geographic region the user has connected from (to enforce legal, region-based content blacklisting), etc. To permit the OA to take these parameters into account, the anticipated values of these parameters can be coded within the task description (specifically, within the task's taint). These anticipated values are then verified at runtime, but only *once* when the task starts running. In practice, this amounts to checking these parameters once per user session, not on every conduit access, which amortizes the cost of the checks.

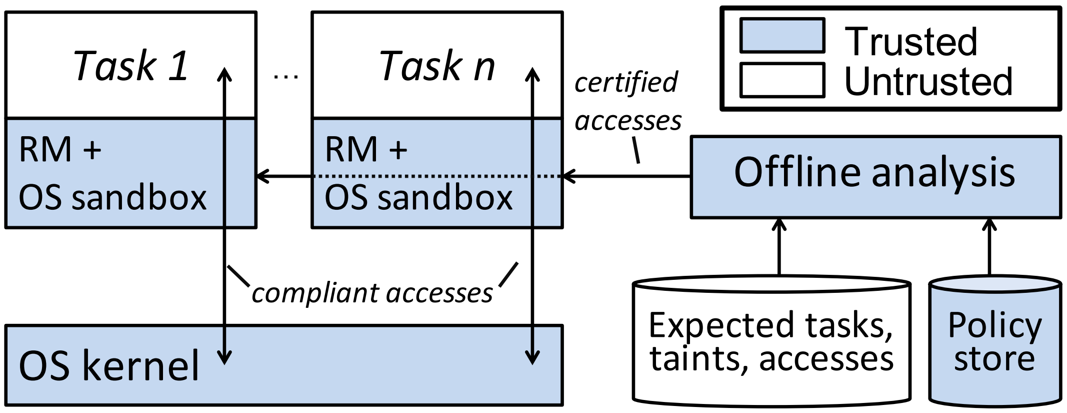


FIGURE 4.2: **S**HAI architecture.

Second, a task's ability to make accesses may depend on meta-data such as friends lists being in a certain state (e.g., Alice can access Bob's friends-only content only while she is Bob's friend). This state may change after the OA has finished, thus (partially) invalidating the OA's analysis. Consequently, the OA must inform the RM of such meta-data dependencies and the RM must track runtime updates to meta-data occurring in the dependencies to avoid policy violations.

Reducing the cost of task reset

The need to reset a user-facing task between sessions of two different users is fundamental to coarse-grained taint tracking and cannot be eliminated entirely. To reduce the cost of this reset significantly, **S**HAI relies on a recent OS primitive called light-weight contexts (*lwC*s) [52] to rollback the state of a user-facing task to a clean state efficiently. *lwC*s support multiple tasks with separate address spaces and file descriptor tables *within* the same process. Resetting an *lwC*'s state resets only the "essential" elements (the memory mappings and open file descriptors) and is faster than re-exec'ing an entire process. This cuts down overheads significantly compared to **T**HOTH.

As an added benefit, the use of *lwC*s also allows implementing the RM itself in a *lwC*, in place of a separate process (as in **T**HOTH). This reduces the cost of interception for the few reads/writes that fault into the RM in **S**HAI from a standard OS context switch to a *lwC* switch, which is cheaper since it does not involve scheduling delays. We describe *lwC*s and their use in **S**HAI in Section 4.3.

4.1.4 Architecture

Figure 4.2 shows SHAI's architecture. SHAI comprises of the offline analysis (OA), and an OS sandbox and the RM. The OA takes as input a list of tasks, what conduits each task is likely to read and write, an estimate of the task's anticipated taint, and the conduits' policies. The OA determines a set of certified accesses for each task. Each task runs in the OS sandbox. The sandbox allows the task's certified accesses (the common case), and faults into the RM for other accesses.

4.1.5 Threat model

Like THOTH, the goal of SHAI is to ensure that policies on ingress conduits are enforced despite bugs in the system's implementation. The concern is inadvertent data leaks, not extraction or stealing of information by malicious adversaries. As such, low-level vulnerabilities (buffer overflows, control flow hijacks, etc.) are not a concern. Implicit flows and side-channels like timing channels would, in principle, be a concern in this setting, but SHAI focuses only on the larger, more prominent risks from explicit leaks of data.

Since SHAI protects only against bugs in application code, the kernel (including its sandboxing mechanism) is trusted. SHAI's integral components—the RM and the OA—are both trusted. Policies on ingress nodes are assumed to represent privacy requirements correctly and all meta-data (e.g., friends lists) on which their interpretation depends is assumed to be accurate.

Policies of internal conduits can be chosen arbitrarily. Getting these wrong can block legitimate data flows in the pipeline, but cannot violate policies of ingress conduits. Any input provided to the OA, with the exception of the policies of conduits, is not trusted. Getting these inputs wrong can only impact performance and/or functionality, not policy enforcement. However, policies of conduits provided to the OA must be the same as those used by the RM.

4.2 SHAI Design

SHAI's design consists primarily of two components—the offline analysis (OA) and a runtime sandbox and reference monitor (RM). We use THOTH's policy language (with very minor extensions) for representing policies. In the following, we first present a running example that we use to illustrate various concepts and that also forms the basis of our evaluation. We then describe the OA and the runtime system.

4.2.1 Example: Search pipeline

Our running example, Sys-E, models the search component of a typical user-facing data-driven system such as a modern social platform. This pipeline is very similar to the pipeline presented earlier in Section 3.4, except for how the search engine transmits the results. We explain (and justify) the difference between Sys-E and the pipeline in Section 3.4 after we review the salient points of the pipeline.

Pipeline tasks. Sys-E indexes a corpus of heterogeneous data consisting of public documents (modeling public content on the WWW), documents private to individual users (modeling content such as emails and individual calendars), and semi-private documents shared among stipulated subsets of users (modeling content such as social media posts that are accessible only to friends or friends of friends). Each piece of content is stored in a separate file. These files are the ingress conduits of Sys-E. The system supports friends lists of users, which are used by the policy enforcement mechanism. The system also has blacklists of documents which should not be visible to users connecting from specific geographic regions to support legal blocking of content.

Indexer. The first task in Sys-E's pipeline is a *data indexer* that builds an index mapping keywords to documents containing those keywords. This task consumes all the content files above and produces the index, which is also stored in files. Note that the data indexer is mostly offline; it only runs periodically.

Search engine. The next task in the pipeline is a *search engine*, which accepts a user query (a set of keywords) over a pipe from a user-facing front-end task (described next), looks up the index, and responds back to the front-end with a *list* of documents that contain those keywords. Technically, the search engine passes open file descriptors for the matching documents over a socket.

Front-end workers. The last part of our pipeline is the *front-end*, which hosts a web server through which remote users interact with Sys-E. For every incoming user connection, Sys-E spawns a new *user-specific worker task*, which authenticates the user (with the user's public key), and then accepts search queries from the user. It forwards each search query to the search engine, then reads each of the matching documents returned by the search engine to extract a snippet, composes all the snippets into a set of "results", personalizes the results using stored preferences of the connected user, and inserts advertisements to generate revenue. It then returns the resulting page to the user. Note that this last part of the pipeline is a not a single task, but consists of a separate task for every connected user.

Relevant policies. The **read** rules of the ingress conduits specify expected confidentiality requirements: Public documents have an all permissive read rule (anyone can read them), Alice's private files have a read rule that allows access to Alice only, and Alice's semi-private files have a read rule that allows access only to Alice and her friends (or friends of friends).

Just as we have explained earlier in Subsection 3.2.2, all the data sources that are used to create the index must allow some declassification (in order for the search engine to produce readable output). In Sys-E, the output of the search engine is a *list of file descriptors of documents* that match a user query. Therefore, we relax the policies of all indexed files to allow such declassification. Specifically, the **declassify** rules of all indexed files allow a complete declassification into any conduit that can only transfer open file descriptors but no other content. The socket from the search engine to a worker task is such a conduit. This allows the search engine to return open file descriptors of matching documents to worker tasks, and allows the pipeline to work as expected. (Technically, the search engine does not have write permission on the socket, and thus, cannot send arbitrary data and can only send file descriptors. An additional check in the kernel, described in Section 4.3, ensures that the worker task can only receive descriptors that it could have opened itself; this prevents a buggy search engine from sending a descriptor for Alice's private file to Bob's worker task.)

The declassification policies of indexed content also have additional clauses for enforcing region-specific censorship. A user's profile (including preferences) has a policy that allows access only to the user. See Subsection 3.4.2 for details.

Data flow from the search engine to the front-ends. The pipeline's tasks, data flows, and associated policies are almost identical to those of the pipeline presented earlier in Section 3.4. The only difference is how the search results are communicated from the search engine to the front-ends. In the pipeline of Section 3.4, the search engine can send only a list of conduit ids to the front-end, and this is enforced by limiting declassification on all searchable content to ONLY_CND_IDS (or ONLY_CND_IDS+ if conduit ids themselves are confidential). While **S**HAI can enforce such typed declassification, this declassification entails intercepting the output of the search engine to ensure that it is indeed a valid list of conduit ids, which would increase **S**HAI's runtime overhead. To overcome this problem, we extend the policy language to express a different typed declassification policy which allows the search engine to return file descriptors of the documents matching a given search query to the front-end. This typed declassification policy can be enforced directly by the underlying OS sandbox, eliminating the need to intercept the search engine output to check for proper declassification.

4.2.2 The offline analysis (OA)

As its name suggests, the offline analysis (OA) is an offline process that runs periodically on the side, not on the critical path of serving requests. The goal of the OA is to check, ahead of time, which conduits each task in the pipeline can read and write. Accesses that check successfully in the OA do not have to be intercepted in the pipeline at runtime, which reduces runtime overhead. To improve efficiency, the OA should be configured to check as many accesses as possible ahead of time. Of course, not all accesses can be checked ahead of time; these accesses are subject to policy checks by the RM as described in Subsection 4.2.3. Accesses to conduits that do not exist when the OA runs, including pipes or sockets, fall in this category. In a properly configured system, those should be the only accesses that are checked at runtime.

The offline analysis inputs

The OA takes the following parameters as inputs:

1. A list of tasks on which to run the OA. If a task's accesses depend on runtime parameters such as the identity of the user the task will serve, a separate instance of the task should be listed for every combination of these parameters.[1]
2. For each task, lists of conduits whose reads and writes by this task have to be checked.
3. The steady-state taint of each task. This is explained below.
4. The policies of all conduits in the system.
5. Any policy-relevant meta-data such as Sys-E's friends lists and region-specific content blacklists.

The taint of a task is a policy that the RM associates to the task at runtime. This policy is always at least as restrictive as the policies of all conduits that the task has read. SHAI enforces this policy on all data that is output by the task and all data that is derived downstream from this output data. The relevance of the taint is that it allows a *local* check to determine if it is safe to allow a task to read a conduit: The read is safe if the task's taint is at least as restrictive as

[1]The identity of the user is not the only possible policy-relevant runtime parameter although, for simplicity, we discuss only this parameter here. Another parameter that our implementation of Sys-E uses is the geographic region from which the user connects; we use this parameter to enforce region-specific legal blacklisting of content.

the conduit's policy since, then, the conduit's policy is guaranteed to be enforced downstream. Input (3) to the OA asks for the runtime steady-state taint of each task.

All inputs (1)–(5) can be determined fairly easily. (1) follows from the schema of the pipeline and, for parameterized tasks, from the possible values of the parameters (e.g., the list of registered users).

The lists in (2) should include as many runtime accesses of the task as possible. These accesses can be determined either by testing, monitoring the production pipeline or manual analysis. For simple pipelines, manual analysis may be straightforward. This works, for instance, for Sys-E: The indexer reads all searchable content and writes the index; the search engine reads the index and indexed content but writes to a socket that is created only at runtime, so the write is irrelevant for the OA; a user's worker task should read only content that is accessible to the user (the user's own private content, content shared by her friends with their friends, public content, etc.) and it writes to a network connection that is also created only at runtime, so this write is also irrelevant for the OA.

(3) can be determined by simple manual analysis, testing or monitoring of the production pipeline. For example, in the Sys-E pipeline, ignoring region-specific censorship for simplicity, the taints are fairly straightforward. (a) Indexer and search engine: Disallow any reads, but eventually allow declassification into a conduit that can only transfer file descriptors, (b) User X's worker task: declassify data only to X.

(4) and (5) should be readily available in the system's meta-data.

SHAI includes a dedicated language to represent (1)–(5); we elide the details of the syntax here, and we refer the interested reader to Appendix B.

The offline analysis operations

The OA checks relevant policies for every conduit read and write mentioned in input (2) and determines which reads and writes are policy compliant and which are not. For simplicity, we first describe the checks assuming that there are no declassification conditions in policies. We then describe the changes needed to handle declassification conditions.

In the absence of declassification conditions, the checks that the OA makes are conceptually straightforward. A task T can *read* a conduit f if f's **declassify** rule (the rule that governs the use of f's data downstream) is at least as *permissive* as T's taint. This ensures that f's data remains protected downstream in accordance with f's policy. Dually, a task T can *write* a conduit f if f's **declassify** rule is at least as *restrictive* as T's taint. This ensures that T's taint is respected on all of T's outputs downstream. For a write, the OA additionally checks that f's **update** rule is satisfied.

When policies have declassification conditions, the check for reads remains unchanged. However, the policy comparison check for writes is more elaborate. The OA first checks if any declassification conditions in T's taint are satisfied. If so, it creates a list of T's updated taints, with one taint for every satisfied declassification condition. If not, it creates a list with only T's current taint. The write is deemed okay if f's **declassify** rule is more restrictive than any of the taints in the list just created.

As an example, suppose that the OA wants to validate a write to conduit f by task T when T's taint is "only Alice can read until the clock time exceeds midnight on December 31, 2019" and f's **declassify** rule is all permissive. (T's taint allows a declassification of Alice's private content at the end of 2019.) In this case, the declassification condition in T's taint is "until the clock time exceeds midnight on December 31, 2019". So, the OA checks whether the clock time is past midnight on December 31, 2019. If this is the case, then T's resulting taint imposes no restrictions, so the write is okay. If this is not the case, then the write is not okay.

These conceptually straightforward checks are more nuanced when they involve metadata that can change over time. Consider the case of Alice's worker task in Sys-E reading a document with the policy "accessible to Bob's friends only". In this case, the policy check above will succeed only if Alice is in Bob's friends list. Suppose that Alice is in Bob's friends list when the OA runs. Now note that, in the future, the validity of this check is *conditional* on Alice remaining in Bob's friends list. If Bob unfriends Alice, this validity is lost.

Consequently, with each access that it successfully validates, the OA also returns a list of conditions on the system state under which the access was validated. We call these conditions the *state conditions* of the access. One general state condition is that the policy of the conduit must be what it was when the OA ran. At runtime, the RM checks these conditions before using the OA's validations. This is explained in more detail in Subsection 4.2.3.

Finally, the validity of the accesses of a task may depend on parameters that can be determined at runtime only. For example, in Sys-E, a worker task serving user X should be able to read only conduits that X is allowed to read, but the identity X will be known only at runtime. In the OA, this is handled by executing the analysis for all possible instances of the parameter (X in this case). Technically, the OA is given a separate instance of the task for every possible value of the parameter. Thus, in Sys-E, there is one instance of the worker task for every registered user—there is a task called "Alice's worker", another called "Bob's worker", etc. The

Algorithm 2 The offline analysis's algorithm

Inputs: (1)–(5) as described in text. In particular, (2) is a list of expected reads of the form (read, T, f) and a list of expected writes of the form (write, T, f).

Output: A list of tuples of the form ([read | write], T, f, $conds$), meaning that T can read or write f if conditions $conds$ hold on the system state.

```
1:  $output \leftarrow \emptyset$
2:  for all (read, T, f) in input (2) do
3:      $pol \leftarrow$ f's policy in input (4)
4:      $taint \leftarrow$ T's taint in input (3)
5:      $(okay, conds) \leftarrow$ isAsRestr($taint$, $pol$.declassify)
6:      if $okay$ then
7:          add (read, T, f, $conds$) to $output$
8:      end if
9:  end for
10: for all (write, T, f) in input (2) do
11:     $pol \leftarrow$ f's policy in input (4)
12:     $taint \leftarrow$ T's taint in input (3)
13:     $(okay1, conds1) \leftarrow$ isAsRestrWithDeclass($pol$.declassify, $taint$)
14:     $(okay2, conds2) \leftarrow$ policyEval($pol$.update)
15:     if $(okay1$ && $okay2)$ then
16:         add (write, T, f, $conds1 \cup conds2$) to $output$
17:     end if
18: end for
19: return $output$
```

specific value of the parameters for a task instance are coded in the taint of the instance. In Sys-E, the taint of Alice's worker is “Only Alice can read downstream”, while that of Bob's worker is “Only Bob can read downstream”. With these precise taints, the OA validates all accesses for the specific instances of the task. At *runtime*, the task must register with the RM as the correct instance, else it won't be allowed to communicate with the connected user. Thus, safety is always maintained.

The offline analysis algorithm. Algorithm 2 summarizes the work of the OA. The algorithm does exactly what is described above. The function isAsRestr($r1$, $r2$) checks that policy rule $r1$ is at least as restrictive as $r2$ and returns a boolean indicating whether this is the case ($okay$) and, if so, what parts of the system state were relevant to this determination (the state conditions, $conds$). The function isAsRestrWithDeclass is similar but it also applies declassification within $r2$. The function policyEval evaluates a policy rule.

All these functions are based on similar functions in **T**HOTH. **T**HOTH uses these functions at runtime, not ahead-of-time. We modified the functions to track which parts of the system state are relevant to the result.

The output of the OA is a list of tuples of the forms (read, T, f, $conds$) and (write, T, f, $conds$) indicating that task T can respectively read or write conduit f if the state conditions $conds$ hold on the system state.

Offline analysis, practical aspects

Predicting runtime context information. As we mentioned earlier, the OA returns a list of state conditions for each certified access. These state conditions are essentially *predictions* on the (future) state of the system at runtime, whose validity is anyway checked at runtime. These predictions are purely performance optimizations: a mis-prediction results in an overhead increase (due to the fall-back to runtime policy evaluation), but it does not affect the correctness of policy enforcement. Hence, it is safe to rely on heuristics for predicting the runtime context information of the system. Next, we discuss a few heuristics that **S**HAI uses.

A key heuristic is that the current state of the system is a good estimate for the future state of the system. This heuristic is particularly useful for the parts of the system state that are changed infrequently. For instance, **S**HAI predicts that the conduits' policies at runtime will be unchanged from those used during OA. Similarly, **S**HAI relies on the existing conduits' content, such as current blacklists and friends lists, as a good estimate for their actual content at runtime. Another heuristic is that previous runtime information contexts are a good estimate of the future contexts of the system. **S**HAI relies on this heuristic, for example, to predict users' geographic locations, where a user who often connects from Germany is predicted to connect form Germany in the future as well.

Besides heuristics, **S**HAI can explore all (or a subset of all) the possible values for a given variable on the system state. In Sys-E for instance, the OA may consider a front-end taint for every user in the system.

OA limitations. There are parts of the system state where there are no heuristics that can provide good predictions and where the space of all possible values is intractable. One example is new content, where the conduit's policy may want to enforce structural integrity on the conduit's content. In these cases, **S**HAI must defer policy checks to the runtime. (No such policies occur in Sys-E.)

Optimizations. It may seem that the total work of the OA is enormous: For every task and every conduit that the task may potentially access, the access should be validated ahead-of-time by the OA for runtime efficiency. In the context of Sys-E, for example, assuming 10 million users and, on average, 1,000 pieces of content accessible to each user, this amounts to 10 billion checks just for the user-specific worker tasks every time the OA runs. This sounds intractable.

In reality, not all these checks are necessary. We describe two obvious optimizations. First, the OA's checks only examine the policies of conduits, not the conduits themselves. Consequently, if a set of conduits share the same policy, then it is safe to run the OA on only one of those conduits and transfer the OA's result to all other conduits in the set. This optimization is quite useful. For example, all of Alice's private content (like her emails) will have the same policy. Similarly, all the uncensored public content on the WWW has the same policy (it is accessible to everyone).

Second, there is no need to include inactive users in the OA often. The OA results remain valid until policies or policy-relevant meta-data change. Since the policies installed on inactive users' content and the policy-relevant meta-data remain unchanged during the inactivity period, it may suffice to include inactive users in the OA sparingly. The OA can also be run on a specific user's content on-demand, e.g., when the system detects that the user has become sufficiently active. Generally, the legitimate accesses that are not part of the OA will not be denied, but those are subject to runtime checks. Therefore, excluding an (inactive) user from the OA will not deny the user's legitimate accesses, but may increase the runtime overhead.

We quantify the cost of the OA on a realistic but simulated workload in Section 4.5.

4.2.3 Runtime monitor and OS sandbox

SHAI's runtime infrastructure consists of two components. First, we rely on an existing OS light-weight capability sandbox[2] to encapsulate every runtime task in the data retrieval system's processing pipeline. The sandbox is configured to allow all accesses that have been validated by the OA without any further interception. Second, a SHAI reference monitor (RM) runs in userspace, isolated within a *lwC*. The RM serves two purposes: It configures the sandbox when a task starts and it validates any accesses that were not validated by the OA ahead of time by making the required policy checks. In the following, we describe these two components somewhat abstractly. Section 4.3 describes a concrete prototype implementation of the RM and the sandbox on FreeBSD.

[2]FreeBSD Capsicum with minimal extensions in our prototype

Task registration

When a new task starts, its access to all conduits is blocked by the OS sandbox; the only thing the task can do is talk to the RM. To get access to conduits, the task must register with the RM by specifying which *previously offline analyzed task* it represents. For example, in Sys-E, the task may register as the indexer, the search engine or user X's worker for any known user X. The RM records the choice and the taint provided during the OA for the specified task.

Next, the RM looks up the last output of the OA for the specified task to determine which accesses for the task have already been validated. For each tuple (read, T, f, $conds$) or (write, T, f, $conds$) in the output, the RM checks the state conditions $conds$, and creates a list of all conduits and permissions for which the conditions hold. It gives this list to the OS sandbox, which subsequently allows the task these accesses directly.

For reasons of efficiency, our prototype implements the checking of state conditions differently. With the exception of state conditions that refer to time and session information, the RM *always maintains* up-to-date lists of each task's valid accesses by tracking changes to meta-data on which state conditions depend (e.g., friends lists of region-specific blacklists in Sys-E), and eagerly re-evaluating state conditions when the meta-data changes. (The validity of state conditions that refer to time and session information is checked at registration, but this is a very simple check and is done once per registration.) As a result, task registration is very fast. The rationale for this implementation choice is straightforward: In online systems like social networks, changes to meta-data like friends lists are far less frequent than task registrations (which happen once per user session), so tying the expensive step of checking state conditions to meta-data changes rather than task registrations results in less overhead.

Registering incorrectly, e.g., registering as the indexer in place of Alice's worker or as Alice's worker in place of Bob's worker, either maliciously or accidentally, cannot cause a policy violation in **S**HAI. However, doing so may cause expected accesses to be denied or more accesses to fault into the RM thus slowing down the task.

Conduit access

After a task has registered with the RM, it can open conduits for reading and writing. Every conduit open call passes through the kernel as usual. If the conduit and the mode (read/write) in which it is being opened were provided to the OS sandbox as a valid access during the task's registration, the kernel just allows the call. This is the fast path and it should apply to most conduit accesses in a properly configured system.

If, on the other hand, the OS sandbox does not know that the specific access is valid, then it transfers control to the RM. The RM then makes the same policy checks that the OA would have made for the corresponding operation (read/write). The only difference is that the RM does not generate any state conditions $conds$; it just checks them immediately. If the checks succeed, the open call is allowed, else it is denied.

Meta-data changes after registration

As explained above, the OS sandbox is informed of a task's pre-validated accesses when the task registers. A relevant question is what to do when a subsequent meta-data change invalidates some of these accesses. There are two choices here: Either the invalidated accesses can be revoked in the OS sandbox or they can be left as is. **S**HAI chooses the latter option since revoking a permission from the sandbox is costly.

This option is also secure since any access that the task does after the invalidation could also have been done before the invalidation to the same effect. An exception to this argument occurs when the read access of a task t to a conduit c is to be invalidated (due to some meta-data change) before the conduit c is updated. In this case, continuing the read access to the conduit c will allow the the task t to obtain the updated content of the conduit c. This is problematic since the task t could not have obtained the updated content had the read access been revoked immediately. To avoid such cases (when they are really a concern), the system should be configured to store updated content in new conduits (e.g., by versioning files). Then, the task t will still have access to the conduit c but not conduit c' that holds the updated content. This choice is compatible with systems like online social networks where existing content is updated relatively infrequently (although fresh content is added quite regularly).

Increasing task taints

In most cases, a task's runtime taint is fixed when the task registers and remains the same throughout its execution. In some cases, however, the task may wish to increase its taint (i.e., make it more restrictive) during its execution. For example, this is necessary if the task wishes to read sensitive content after writing to a public conduit. In this case, the task must start with a public taint and acquire the taint of the sensitive content afterward.

SHAI allows a task to increase its taint at runtime as follows. At any point, a running task may re-register as a another task (previously analyzed by OA) whose taint is higher than the task's current taint. **S**HAI checks that the new taint is at least as restrictive as the current taint, and also checks that the policies of any conduits to which the task has open write handles are more restrictive than the task's new taint. These checks are necessary to prevent leaks of data that the task reads under the new taint. If this check fails, the re-registration is disallowed.

On the other hand, reducing a task's taint at runtime is not safe as this allows the task to leak previously read information (which are subject to a more restrictive taint). Therefore, SHAI, like all other coarse-grained taint tracking systems, disallows reducing a task's taint.

Runtime interception cost, an example

The overall cost of runtime interception in SHAI is generally very low. For interactive pipelines such as Sys-E's RM interception happens only a few times per user session (not per user query). For instance, in Sys-E, only four RM interceptions are needed per session:

1. When the worker accepts the user's connection.
2. When the session is authenticated. Users authenticate directly with the RM.
3. To register the worker as a task for the connected user. This interception also validates the policies on all conduits the task has write handles for, including the outgoing user connection and the pipe that connects the worker to the search engine.
4. At the end of the session, to reset the worker task to a clean state for the next user.

As we mentioned earlier, the RM intercepts accesses that were not certified by the OA. Such interceptions should be few in a properly configured pipeline.

4.3 SHAI Prototype

Our SHAI prototype runs on FreeBSD and relies on FreeBSD's kernel capability support (Capsicum) [89] and light-weight contexts (*lwC*s) [52] for sandboxing and isolation, respectively. We discussed both in Chapter 2.

Figure 4.3 depicts the runtime architecture of our SHAI prototype. SHAI maps tasks one-one to Capsicum-sandboxed *lwC*s. The RM also runs inside a privileged (unsandboxed) *lwC*. The kernel is configured to redirect any syscall outside a task's capability set to the RM *lwC*. As compared to a design that uses processes (rather than *lwC*s) for the same purposes, this design allows for faster switching between tasks and the RM, and for faster resetting of tainted worker tasks at the end of user sessions by avoiding scheduler delays.

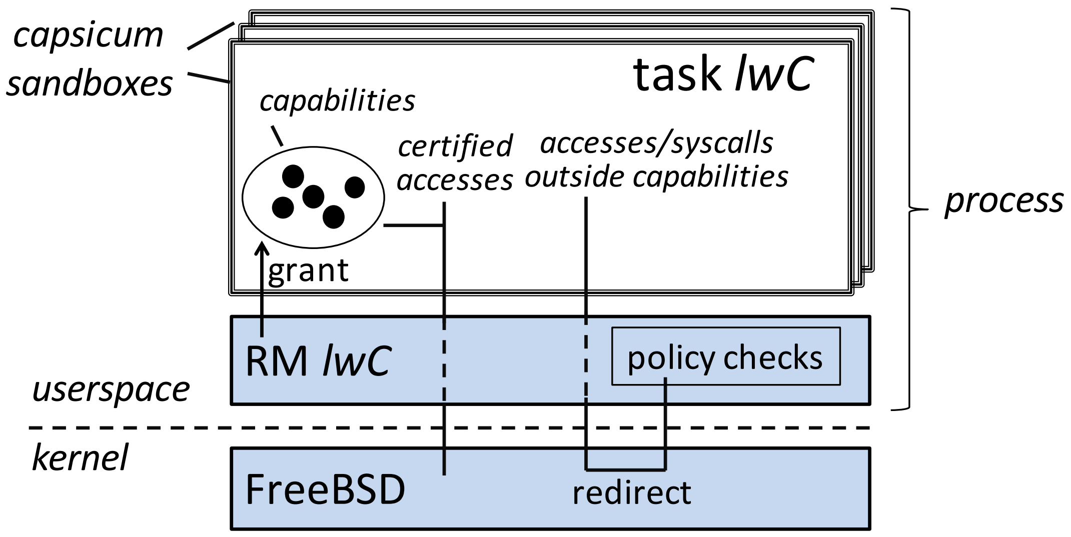


FIGURE 4.3: SHAI prototype.

The RM in our SHAI's prototype is around 21,000 lines of C code. Additionally, we have implemented a dynamically-linked library of 1080 lines of C code, which userspace applications can use to call SHAI's API. SHAI's API extends THOTH's API by including only one more API call (REGISTER(T_ID)) to register the calling task as a previously offline analyzed task.

Application life cycle. An application is loaded with a script that first initializes the RM *lwC* in each process. Next, it initializes application tasks in separate *lwC*s, confining them with Capsicum's capability mode. Then, the RM is invoked to register each application task, giving it the capabilities to access anything that was already certified by the OA and whose state conditions hold. Depending on the type of a conduit, the capability to access it takes different forms:

- Files: The task is given *Capsicum* capabilities to a set of directories that contain hard links to all files that should be accessible to the task. These directories and the hard links are created offline at the end of the OA and kept up-to-date by the RM as state conditions change.
- Key-value (KV) tuples: For these, the RM relies on KV filters. The RM opens a socket to the KV store and installs a KV filter that limits access to only those tuples that are accessible to the task. It then gives this open socket to the task.

During its execution, an application task makes most conduit accesses directly using the capabilities described above. For accesses that are beyond these capabilities, the application faults into the RM, which makes policy checks. For instance, the RM intercepts all socket establishments (which are typically a few per task in Sys-E) and performs policy checks (i.e., can the application declassify into a socket given its taint?). If the checks succeed, the RM permits establishing the socket. Afterwards, reads and writes to the socket are not intercepted by the RM.

Capsicum modifications. To support Sys-E and other similar search-based pipelines, we made two modifications to Capsicum.

- We modified Capsicum to allow a socket without read and write permissions to be used to transfer open file descriptors but not data. In Sys-E, such a socket is used by the search engine to return file descriptors for documents matching the user query to the front-end's worker task. Since data transfers on such socket are forbidden, even a buggy search engine cannot accidentally send private data to the worker.
- We modified Capsicum to allow a task in capability mode to transfer file descriptors to another task in capability mode only if the receiving task already has access capabilities on all conduits referenced by the file descriptors. With this feature in place, Capsicum prevents a buggy search engine from transferring a descriptor for Bob's private file to a front-end worker task connected to Alice. To implement this feature, we modified Capsicum to maintain every task's capabilities in a binary lookup tree. When a source capability mode task transfers a file descriptor to a destination capability mode task, Capsicum looks up the binary tree of the destination for a capability to the conduit referenced by the descriptor. This lookup's complexity is logarithmic in the number of distinct capabilities the task has. In Sys-E, only the front-end tasks receive file descriptors and these tasks have very few capabilities, so the lookup is very fast.

4.4 Policy-compliant data retrieval with SHAI

We have implemented the data retrieval system Sys-E described in 4.2.1. We instantiated this system with Apache Lucene as the search engine [3]. Our prototype supports replication, index sharding, and features such as personalization and advertisements in the same way that was described in Section 3.4. In this section, we describe our experience using SHAI to enforce policies in Sys-E.

In our **S**HAI prototype, applications run in Capsicum's capability mode. Since system calls that access the global file system namespace, such as the OPEN system call, are denied categorically in Capsicum's capability mode, we changed Sys-E's applications to use the capability-aware variants of these system calls. For instance, applications may use OPENAT instead of OPEN. In capability mode, OPENAT system calls are allowed subject to valid capabilities. We found that OPENAT is faster than OPEN.[3] Hence, we carried this modification to the baseline configuration of Sys-E, regardless of whether tasks run in Capsicum or not. These modifications were small:

- For Java applications (the Lucene indexer and Lucene searcher), we modified less than 50 lines of code of Lucene's codebase to call into a 550 lines of C code in a JNI library that performs capability-based I/O.
- For C applications (the front-end), we replaced all fopen/open calls with the fopenat/openat variants. This amounted to less than 20 lines of code.

Next, we provide commentary on the code changes required to adapt the three core components of Sys-E (indexer, search, and front-ends) to **S**HAI. None of these code changes affect the correctness of policy enforcement but rather make Sys-E's data flows comply with the policies in place.

Indexer Task. We have introduced two changes to the Apache Lucene indexer.

- **Register as the indexer task.** The indexer must register with **S**HAI's RM in order to acquire access capabilities on the searchable content.
- **Set policies appropriately on the index files.** Similar to **T**HOTH, the indexer in **S**HAI attaches the index policy to newly created index files. (Otherwise, file creation will not be allowed if the associated policy is not at least as restrictive as the indexer's taint).

Search Task. We have introduced two changes to the Apache Lucene searcher in order for the search task to function in **S**HAI, in capability mode.

- **Register as the search task.** The search task registers with **S**HAI's RM to acquire read access capabilities on *(i)* the index files, *(ii)* the query pipes (to read users' search queries), and *(iii)* the searchable content (to open file descriptors for the conduits that match a given query).

[3]OPEN needs to resolve the *full* path name of a conduit, where OPENAT uses the directory cache to resolve the conduit's name which is faster.

- **Produce results as a list of FDs.** In order to send the query results, the search task establishes socket connections to the front-end tasks (one connection per front-end task) at runtime. For a given query, the search process creates file descriptors for the conduits that match the query and passes those file descriptors to the front-end task over the socket using the SEND_MSG system call. Capsicum allows the transfer of an open file descriptor between two capability-mode tasks (e.g., the search and the front-end tasks) as long as the destination already has a capability on the conduit referenced by the file descriptor. However, if the front-end task *does not* have capabilities on the conduits referenced by the file descriptors[4], Capsicum does not allow the SEND_MSG system call and directs it to SHAI's reference monitor for policy evaluation at runtime. In order to help with policy evaluation, the search process may provide evaluation hints (e.g., caching entries in a friends list). Providing evaluation hints is analogous to what the front-end in the THOTH-compliant search pipeline does (see Subsection 3.4.2).

Front-end Tasks. As mentioned earlier, the front-end tasks are not part of Apache Lucene's codebase. Our implementation mimics a simple web server that accepts users' connections, submits users' queries to the search tasks over pipes, consumes the results as a list of open file descriptors, and prepares and sends search snippets to users. We highlight two core functionalities that are required for the front-end tasks to function with SHAI.

- **Register as a user task.** The front-end task has worker *lwCs*. Each worker *lwC* accepts an incoming user connection. The user can authenticate directly with SHAI's RM, and the task may attempt to register with the RM as the connected user's task to acquire the already certified accesses.
- **Task reset.** When a connected user terminates her session, the front-end destroys the session's *lwC* and creates a pristine *lwC* that can accept a new incoming connection.

4.5 SHAI Evaluation

In this section, we present results of an experimental evaluation of our SHAI prototype. In particular, we measure the overhead of policy enforcement in our prototype data retrieval system Sys-E.

[4]This may happen, for instance, due to mispredictions during offline analysis.

All experiments were performed on Dell R410 servers, each with 2 Intel Xeon X5650 2.66 GHz 6 core CPUs, 48GB main memory, running FreeBSD 11.0 (x86-64) with support for light-weight contexts (*lwC*) [52] and Lucene version 4.7. The prototype uses OpenSSL v1.0.2h. The servers are connected to Cisco Nexus 7018 switches with 1Gbit Ethernet links, which offer enough network bandwidth for all our experiments. Each server has a 1TB Seagate ST31000424SS disk formatted under UFS, which contains the OS installation and a 258GB static snapshot of English language Wikipedia articles from 2008 [91].

Experimental setup. In the following experiments, we compare the performance of **S**HAI to two systems; *(i)* a system that does not enforce policies (**B**ASELINE), and *(ii)* a system that enforces policies via *pure* dynamic analysis (**D**YNAMIC). **D**YNAMIC is very similar to **T**HOTH. In the following, we describe **D**YNAMIC in detail and outline how it differs from **T**HOTH.

In **D**YNAMIC, each task has a *current* taint, which represents the combined policies of all the conduits the task has read. A task's taint can become more restrictive as the task reads more conduits. To enforce policy, a task's writes must *(i)* satisfy the **update** rule of the destination conduit's policy, and *(ii)* satisfy the declassification conditions of the task's current taint. **D**YNAMIC is very similar to **T**HOTH; for fair comparison, our **D**YNAMIC implementation, like **S**HAI, takes advantage of *lwCs* and Capsicum for efficient in-process isolation and sandboxing. This yields better performance than the original **T**HOTH prototype, which isolates each user session and the reference monitor in a separate process.

A process in **D**YNAMIC, like **S**HAI, can have multiple Capsicum-sandboxed user *lwCs* (each terminates a user connection), and a privileged monitor *lwC*. Conceptually, the RM intercepts all conduit open calls and writes to perform taint tracking and dynamic policy checks. As in **T**HOTH, we optimize taint tracking by not invoking the RM during open calls and instead logging such calls in the kernel. During a write, the RM is invoked, it checks the open call trace to update the task taints and then performs the policy check for the write. To summarize, **D**YNAMIC and **S**HAI are identical architecturally: A process has multiple Capsicum-sandboxed user *lwCs* and a privileged monitor *lwC*. Both systems also enforce the same policies. However, unlike **S**HAI, which pushes most policy evaluation overhead to the offline analysis, **D**YNAMIC performs pure dynamic information flow control: the underlying kernel intercepts I/O and directs it to the reference monitor, which in turn tracks taint and performs policy evaluation at runtime.

4.5.1 Search throughput

First, we measure **S**HAI's overhead on search throughput. We drive the experiment with the following workload. We simulate a population of 40,000 users. Each user is assigned a friend list consisting of 100 randomly chosen other users, subject to the constraint that the friendship relationship is symmetric. Each document in the Wikipedia corpus is assigned either a public, private, or friends-only policy in the proportion 50/30/20%, respectively. Private and friends-only documents are assigned to a user picked uniformly at random from the population. A total of 1.1% of the corpus is censored in some region. A censored document's policy allows declassification to an external user only if the destination's blacklist file does not blacklist the document.

In this experiment, 24 concurrent users issue queries in parallel. We use query strings based on the popularity of Wikipedia page accesses during one hour on April 1, 2012 [90]. Specifically, we search for the titles of the top 20K visited articles and assign each of the queries randomly to one of the users. User sessions run for lengths 1, 2, 4, 8, 16, or 32 queries. Additionally, we report the throughput when users maintain their sessions for the duration of the experiment (20k queries).

In our setup, two server machines execute a Lucene instance with different index shards. The front-end submits a search request to one Lucene instance, which in turn forwards the request to the other instance and merges the results from both shards. To maximize the performance of the baseline and fully expose the policy enforcement overheads, the index shards and parts of the corpus relevant to our query stream are pre-loaded into the servers' main memory caches, resulting in a CPU-bound workload. To ensure load balance, we partitioned the index into two shards of 22GB and 33GB, chosen to achieve approximately equal query throughput.

Table 4.1 shows the average throughput over 40 runs of 20K queries each, for **B**ASELINE, **D**YNAMIC, and **S**HAI. The standard deviation over the 40 runs was below 0.87% across all configurations.

The key result is that **S**HAI's offline analysis reduces the runtime enforcement overhead to near zero for sufficiently long session lengths (0.1% and 0.02% at 16 and 20k queries per session, respectively). The **D**YNAMIC system, which relies on pure runtime enforcement but is otherwise equivalent, has a runtime overhead of approximately 2.5% for large session lengths.[5] Even for short session lengths, **S**HAI's runtime overhead is substantially lower than

[5]A 2.5% overhead may seem small; but increasing the peak capacity of a large datacenter by 2.5% to account for it has a substantial cost!

	BASELINE	DYNAMIC		SHAI	
Q/S	Avg.	Avg.	overhead	Avg.	overhead
1	309.36	287.04	7.21%	294.70	4.74%
2	313.90	298.29	4.97%	305.60	2.64%
4	316.49	304.58	3.76%	312.54	1.25%
8	317.60	308.07	3.00%	316.17	0.45%
16	318.66	309.27	2.95%	318.34	0.10%
32	318.95	310.11	2.77%	318.64	0.10%
20k	319.13	311.10	2.52%	319.08	0.02%

TABLE 4.1: Average search throughput in queries per second. Standard deviation was less than 0.87% from the average in all cases. First column indicates the session length (queries per session – Q/S).

DYNAMIC's.

In **D**YNAMIC and **S**HAI, the front-end creates a new *lwC* for every incoming user session. In **S**HAI, the monitor *lwC* additionally performs the required runtime checks associated with the connected user's taint before granting access capabilities. The overheads of setting up new sessions (creating *lwCs* and performing runtime checks) dominate the policy enforcement overhead for short sessions. At one query per session, **D**YNAMIC incurs a 7.21% overhead, whereas **S**HAI incurs 4.74%. Here, **S**HAI outperforms **D**YNAMIC since *(i)* **S**HAI performs fewer (runtime) checks compared to **D**YNAMIC's full policy evaluation for all documents accessed per query, and *(ii)* **D**YNAMIC tracks the search engine's taint and intercepts its writes to evaluate them against the search engine's current taint, whereas the search engine's accesses within its capability set are not intercepted in **S**HAI. Furthermore, as the session length increases, the cost of **S**HAI's per-session setup costs and runtime checks amortize over the session's queries, whereas **D**YNAMIC performs full policy evaluation for each query.

At session length 20k, **S**HAI incurs 0.02% overhead, significantly better than **D**YNAMIC's 2.52% overhead. **S**HAI's remaining runtime overhead is due to the kernel's capability checks; in particular, when the search engine attempts to send file descriptors corresponding to the search results, the kernel checks that the front-end has existing access capabilities for these descriptors. This check is efficient; its runtime complexity is logarithmic in the number of distinct (directory) capabilities the receiving front-end has. In our prototype, a front-end that satisfies runtime checks acquires few directory capabilities[6], making the check light-weight.

[6]One on the connected user's hard links directory, another on named pipes to submit queries to the search engine, and three on directories with public documents.

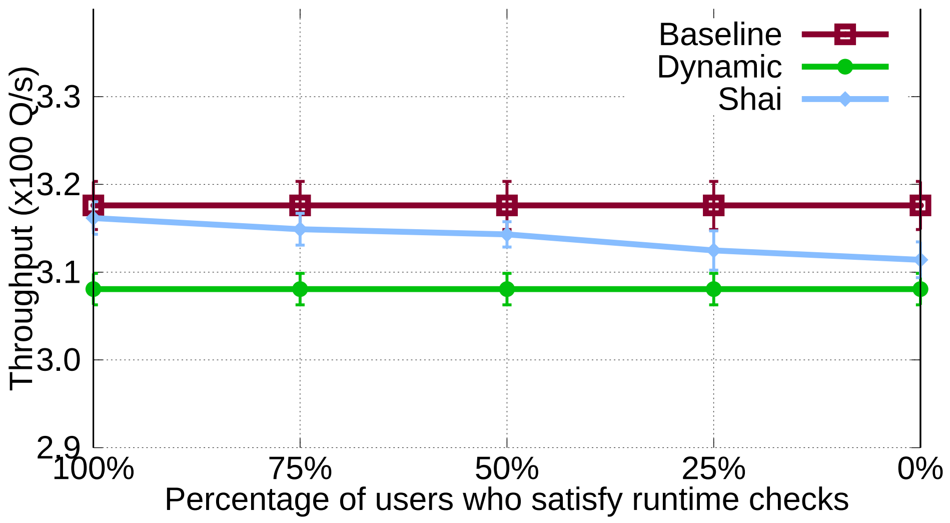


FIGURE 4.4: Average search throughput in queries per second of 24 concurrent users, with sessions of length 8 queries. We included BASELINE and DYNAMIC performance (upper and lower lines, respectively) for reference. Error bars show standard deviation.

In the previous experiment, all users connected from the regions that were anticipated during the offline analysis. When a user connects from a different (not anticipated) region, the user's accesses will be subject to runtime checks (as the user's task will not be able to acquire access capabilities). To quantify the overhead of runtime checks required when runtime conditions deviate from those anticipated, we next perform an experiment in which we vary the proportion of users who connect from regions different from those assumed during offline analysis. Figure 4.4 shows the average throughput over 40 runs of 20K queries each. The error bars indicate the standard deviation over the 40 runs, which was less than 0.72% in all cases. We report the average throughput for sessions of length 8 queries, but the following conclusions regarding the relative overheads of SHAI and DYNAMIC hold across all session lengths.

With 100% of users connecting from the expected region (i.e., all user sessions satisfy the runtime checks associated with their taint), SHAI performs 316.17 Q/s (0.45% overhead over BASELINE), as in Table 4.1. As the proportion of users who connect from their expected regions decreases, SHAI's performance declines approximately linearly and approaches that of DYNAMIC. We note that as SHAI's performance decreases due to mispredictions, SHAI's overhead is strictly lower than DYNAMIC's. Even when all users connect from unexpected regions, DYNAMIC incurs more overhead than SHAI because it intercepts the search engine's writes to evaluate them against the search engine's current taint. More generally, SHAI's benefits decline gracefully with the accuracy and the freshness of the offline analysis. For instance, SHAI's throughput declines approximately linearly as the proportion of accesses to new content (since the last offline analysis) increases.

4.5.2 Scaling search throughput

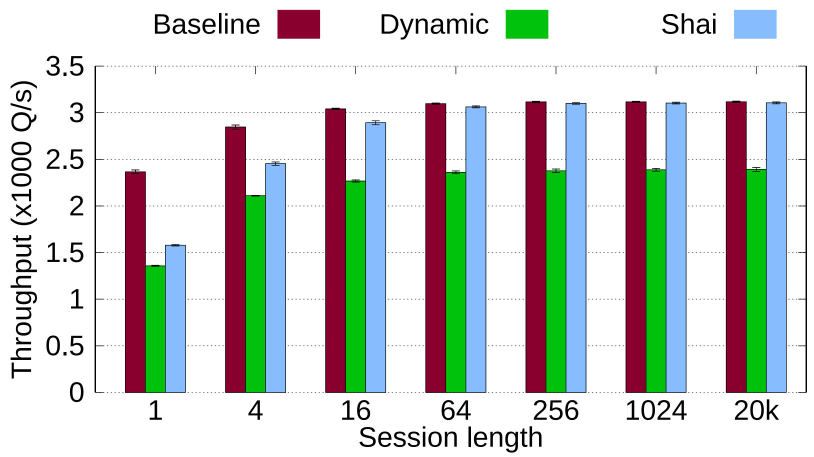
The throughput of a single Lucene search engine is relatively modest, which raises the question of how much overhead SHAI might impose on a much faster system. In the next set of experiments, we study SHAI's overhead in a replicated search engine configuration, and in a configuration with a hypothetical search engine that has much higher throughput than Lucene.

Replication

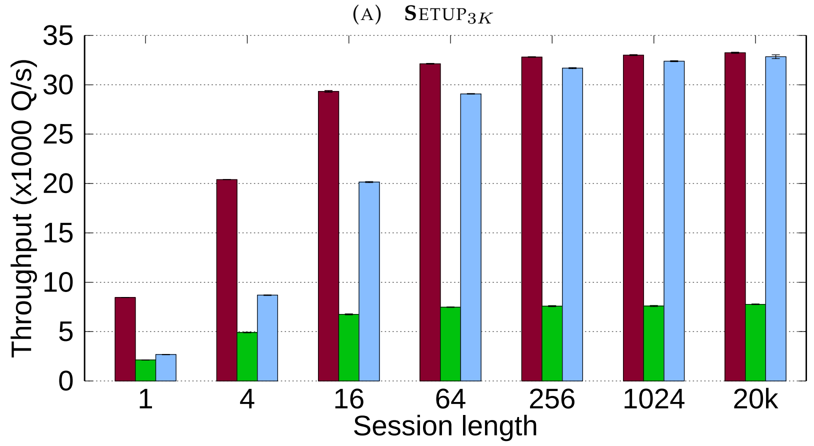
We performed the throughput experiment on a replicated setup. In this experiment, four server machines execute Lucene instances, where each index shard is replicated on two servers. A front-end submits a search request to a lightly loaded Lucene instance, which in turn forwards the request to another lightly loaded instance processing the other shard and merges the results from both shards. Here, 48 users issue queries in parallel, users maintain their sessions for the duration of the experiment, and we measured the average throughput over 40 runs, each 20K queries. BASELINE, DYNAMIC, and SHAI all achieved an average throughput of almost exactly twice (within 0.152%) the respective throughput reported in Table 4.1 at session length 20k. This shows that SHAI (like DYNAMIC) scales linearly as the search engine is replicated.

Hypothetical fast search

To study SHAI's overhead in data retrieval systems that serve tens of thousands of search requests per second, we replaced the Lucene search engine with (a hypothetical, extremely fast) one that picks results randomly from the set of documents accessible by the user who issues the query. We measure SHAI's overhead over *(a)* a dummy search engine that performs over 3K Q/s (SETUP$_{3K}$), and *(b)* a dummy search engine that performs over 30K Q/s (SETUP$_{30K}$). The dummy search engine busy waits to consume a fixed number of CPU cycles in SETUP$_{3K}$ before returning the search results, whereas it returns the results immediately without busy waiting in SETUP$_{30K}$. Note that SETUP$_{30K}$ represents an extreme situation, shown here only



(A) SETUP$_{3K}$



(B) SETUP$_{30K}$

FIGURE 4.5: Search throughput in (Q/s) of 56 concurrent users, at different session lengths. Error bars show standard deviation.

to fully expose SHAI's overheads; we do not expect any realistic search engine to attain such high per-node throughput.

In this experiment, a total of 56 concurrent users issue queries in parallel to two server machines running the dummy search engine. User sessions run for lengths 1, 4, 16, 64, 256, 1024, and 20k queries. Figure 4.5 shows the average throughput at the different session lengths for SETUP$_{3K}$ and SETUP$_{30K}$ (Figures 4.5a and 4.5b, respectively). We report the average throughput over 10 runs, each of length 30 seconds. Error bars show the standard deviation across the 10 runs, which was below 0.9% in all cases.

At small session lengths, both SHAI and DYNAMIC have high overheads due to the cost of creating *lwCs* to isolate user sessions. As the session length increases, the cost of session creation amortizes across queries in **S**HAI; the overheads are only 0.37% and 1.2% at session length 20k, in 4.5a and 4.5b respectively. These overheads are due to checking, at a high rate, that the front-ends have existing capabilities over the transferred file descriptors. On the other hand, **D**YNAMIC does not scale beyond 2.39K and 7.76K Q/s in 4.5a and 4.5b, because intercepting I/O to perform policy evaluation at runtime limits performance. This result shows that **S**HAI can maintain low overhead even in very high-performance data retrieval systems while **D**YNAMIC cannot.

4.5.3 Search latency

We next measure **S**HAI's overhead on query latency. For this experiment, a user issues one query at a time and waits until it receives a result before issuing another query. User sessions run for lengths 1, 2, 4, 8, 16, 32 or 4k queries.

Q/S	**B**ASELINE	**D**YNAMIC	**S**HAI$_{mispredict}$	**S**HAI
1	36.013	38.071	37.314	36.345
2	36.007	37.843	37.122	36.108
4	36.005	37.742	37.084	36.037
8	35.981	37.719	37.021	36.008
16	35.939	37.657	36.916	35.955
32	35.928	37.599	36.904	35.941
4k	35.905	37.597	36.913	35.960

TABLE 4.2: Average query latency (ms). Standard deviation was less than 0.8% in all cases. The first column indicates the session length (queries per session – Q/S).

Table 4.2 shows the average query latency across 5 runs of 4K queries. Since **S**HAI's overhead relies on satisfying the runtime checks necessary to acquire capabilities, we report **S**HAI's performance when *(i)* the user logs in from a geographic location different than the region used during offline analysis (Table 4.2 column 4: **S**HAI$_{mispredict}$), and when *(ii)* the user logs in from the geographic location that is used in the offline analysis (Table 4.2 column 5).

SHAI's overhead on query latency is very low (at most 0.34ms). Also, **S**HAI$_{mispredict}$ (when the user fails to satisfy the runtime checks to acquire capabilities) achieves better performance than **D**YNAMIC. The performance difference is due to tracking the taint and intercepting the search engine writes in **D**YNAMIC, whereas the search engine's accesses within its capability set are not intercepted in **S**HAI$_{mispredict}$.

4.5.4 Offline analysis

Next, we measure the cost of running the offline analysis over the expected data flows within the data retrieval system. The runtime of the analysis depends on the number of tasks, the number of expected accesses and relevant policies of each task, and the number of accesses certified (each certified access may require creating a hard link in the task's capability directory). We limit the analysis to a single CPU. The analysis can be sped up by using more CPUs since its computation is embarrassingly parallel (except when creating hard links within the same directory).

Indexer and search engine flows

The analysis takes under 2 seconds to process the flows of the search engine and the indexer tasks on the entire Wikipedia corpus ($\sim$14.5 million documents subject to $\sim$80K different policies). Here, the searchable documents' policies permit read to the indexer and the search engine, and the analysis grants both a single capability for the top level directory of searchable documents.

Users flows

Next, we measure the analysis time and storage requirements for the users of the data retrieval system. For this experiment, we assume a fixed default geographic location for every user. For each user, the offline analysis checks the front-end's accesses of all public documents, the user's private documents, and the friends-only documents of the user's friends.

We ran the OA on accesses of 100 users picked at random from the population. Processing the accesses took 90.5 seconds per user on average. This can be optimized using a faster storage medium since most of this time (96.1%) was spent waiting for the magnetic disk to record hard links for access capabilities. To quantify potential speed-up when using a ramdisk, we ran the offline analysis for the same 100 users on a Dell R640 server machine with 385GB main memory, and limited the analysis to use only one core of its Xeon Gold 6142 2.60GHz CPUs. This server machine has enough memory to store the entire Wikipedia corpus in ramdisk, allowing us to create hard links in ramdisk too. Using the ramdisk to store hard links, processing each user took under 1.2 seconds on average (20% of which was spent creating hard links).

Each user's access capabilities consumed 12.9MB of disk space to store 145.8K hard links on average. Tasks' taints and state conditions consumed less than 11MB of disk space for all 100 users combined.

4.5.5 Indexing

Finally, we measure the overhead of policy enforcement on the index computation. We run the Lucene indexer over the entire 258GB snapshot of the English Wikipedia. The resulting index is 54GB in size. Table 4.3 shows the average indexing time in minutes across 3 runs. The standard deviation was less than 2% in all cases.

	Average	Overhead
BASELINE	652.27	
DYNAMIC	672.02	3.02%
SHAI	656.16	0.59%

TABLE 4.3: Indexing time in minutes.

Enforcing policies with SHAI during indexing incurs a runtime overhead of 0.59%, which is significantly lower than DYNAMIC's 3.02%. SHAI's overhead is due to the fact that the indexer creates many new files, and all these file creations must be intercepted to ensure that output has appropriate policy given the indexer task's taint. Policy enforcement in DYNAMIC additionally intercepts the indexer's writes to the index files and tracks the indexer's taint.

Since indexing is a relatively infrequent operation in a search pipeline, we believe that a runtime overhead of 0.59% is acceptable. However, in other systems where frequent file creation occurs on the critical path, runtime interception of file creation could be avoided as follows. Using an appropriate Capsicum capability, we can restrict file creation to a specific directory with an appropriate policy. All files created in this directory implicitly inherit this policy. The offline analysis can check upfront that the task creating the files can write to any file with this policy.

4.5.6 Fault-injection tests

To double-check SHAI's ability to stop data leaks, we ran the fault injection tests that we used in THOTH's evaluation in 3.5.3. In all tests, SHAI stopped all injected data leaks.

Moreover, we injected faults in the search engine to *(i)* produce data (rather than open file descriptors) on a socket connected to a front-end, and to *(ii)* send an open file descriptor for Alice's private file to Bob's front-end task. SHAI stopped the leaks in both cases.

4.6 SHAI Conclusion

SHAI shows that it is possible to enforce data-specific flow policies in data retrieval systems with near-zero runtime overhead in the common case. SHAI relies on a combination of an offline flow analysis, session-level binding of runtime variables, and light-weight runtime monitoring using an OS capability sandbox to achieve this goal. The key insight behind SHAI is to push as much work as possible to the offline analysis, often relying on anticipated values of runtime parameters, and to use efficient OS techniques (light-weight contexts and Capsicum capabilities) to minimize runtime overhead. This combination keeps SHAI's overheads low, even when the system throughput is high.

5 Related Work

This chapter relates the work presented in this thesis to prior work.

5.1 Data Retrieval Policy Compliance

Grok. Grok [75] is a privacy compliance tool that is deployed in the back-end data analysis pipelines of the Bing search engine. The primary goal for Grok is to allow automated privacy compliance checking in a large-scale system. Towards this end, Grok uses static analysis (with heuristics) over programs and job logs to construct a data flow graph between processes, data stores, and entities (users or functional teams). Grok uses automatic inference and selective manual verification by developers to assign *attributes* to the graph nodes. Grok attributes are domain-specific labels and represent the intended policy restrictions. Grok policies, written in a language called Legalease, specify allowed data flows on attributes. To detect privacy violations, the Grok policies are checked against the labeled data flow graph. Since the labels may be incorrect as they are derived using unsafe heuristics, the detected violations may have false positives and false negatives, which must be resolved manually. Nonetheless, Grok demonstrates that automated policy compliance checking can scale to actual production pipelines.

Grok and the systems presented in this thesis differ in target policies and enforcement techniques. Grok focuses on the privacy policies which are relevant to the search engine's back-end and which apply generically to all data of a specific type (e.g., full IP address cannot be used for advertising). Therefore, Legalease policies and attributes apply at the granularity of data types (e.g., table columns for IP addresses, clicks, and user accounts). However, this granularity cannot express individual (data- and user-specific) policies which are particularly relevant for the user-facing tasks. Therefore, unlike the policy language we presented in this thesis, Legalease cannot express policies such as private and friends-only data policies ("Carol's e-mail is private to Carol only" and "Alice's blog post is available to Alice and her friends"). Legalease also does not support content-dependent policies and cannot express mandatory access logging (e.g., when a log entry must exist before accessing sensitive content), censorship (e.g., when a data item is censored in a specific jurisdiction), and typed declassification (which allows the declassification of data in specific forms/types) policies.

Due to the different target policies, the systems developed in this thesis use different enforcement techniques than Grok's. Grok establishes policy compliance with a fast static analysis which is a good match for policies specified at the granularity of data types. Pure static analysis-based policy enforcement, however, cannot enforce individual policies. (This is primarily due to *(a)* precision loss when a given program variable is tainted with different individual policies over time, and *(b)* lack of policy-relevant runtime information such as connected users' identity and their geographic location.) Instead of static analysis, **T**HOTH and **S**HAI rely on runtime monitoring and on offline (flow) analysis and light-weight runtime checks, respectively.

Since Grok relies on static program analysis, it is language-dependent. It works on languages such as Hive, Dremel, and Scope where jobs are a sequence of SQL-like expressions. Moreover, Grok may have false negatives and false positives (since the static analysis uses unsafe heuristics). **T**HOTH and **S**HAI do not depend on the language or the runtime (since they rely on OS primitives, below applications), and offer safe enforcement (no false negatives). On the other hand, Grok has zero runtime overhead, while **T**HOTH and **S**HAI incur small runtime overhead.

Invariant Detector (IVD). IVD [56] is a system to automatically infer authorization rules in Facebook's online social network. At a high level, IVD learns likely authorization rules from normal data manipulation patterns. These authorization rules are checked at runtime against user requests, preventing bugs due to missing authorization checks by blocking requests that would otherwise break such invariants.

The systems developed in this thesis prevent application bugs from violating data use policies, whereas the IVD system prevents vulnerabilities due to missing or incorrect authorization checks. These are different goals. On one hand, the goal of preventing policy violations is to protect the integrity and confidentiality of data. Ensuring that the applications themselves do not deviate from their expected behaviour is out of scope. For instance, the systems developed in this thesis do not protect applications against a (malicious) user invoking debugging or under-development features that are not meant for external users —as long as the invocation does not violate data policies. On the other hand, IVD's goal of preventing vulnerabilities due to incorrect authorization checks is to ensure that systems and applications do not deviate from their intended behaviour. For instance, IVD blocks a user request if the corresponding application actions deviate from normal, expected behaviour —regardless of being compliant with data policies. In this regard, the systems developed in this thesis and IVD are orthogonal.

Despite having different goals, IVD's authorization rules can implicitly express data-specific *write* access control policies, such as "only Alice's friends can post to Alice's profile". IVD, however, does not enforce access control on reads and, thus, cannot enforce data confidentiality, while **T**HOTH and **S**HAI can enforce rich integrity, confidentiality, and declassification policies.

5.2 Cloud Policy Compliance

Maniatis et al. [54] outline a vision, architecture and challenges for data protection in the cloud using secure data capsules. The compliance systems presented in this thesis can be viewed as a realization of that vision in the context of a data retrieval system, and contribute the design of a policy language, different enforcement techniques, and experimental evaluation.

Secure Data Preservers (SDaPs) [44] are software components that mediate access to data according to a user-provided policy. SDaPs are suitable only for web services that interact with user data through simple, narrow interfaces, and do not require direct access to users' raw data. This is in contrast to **T**HOTH and **S**HAI which enforce policies over standard OS interfaces to users' raw data.

LoNet [43] enforces data-use policies at the VM-level. Declassification requires trusted application code, and interception is limited to file I/O using FUSE, which results in high runtime overhead. This is in contrast to **T**HOTH and **S**HAI, which enforce policies at the process boundary, require no trusted applications for declassification (since declassification conditions are encoded in the policies), and incur low runtime overhead (optimized kernel-level interception in **T**HOTH, and light-weight runtime monitoring via OS-sandboxes in **S**HAI).

5.3 Information Flow Control (IFC)

Numerous systems restrict a program's data flow to enforce security policies, either in the programming language (JFlow/Jif [62, 63] for Java, Flow Caml [69] for OCaml, LIO [80] for Haskell, and UrFlow [21] for Ur/Web [22]), in the language runtime (RESIN [98] for PHP and Python, Nemesis [24] for PHP, and Aeolus [20] for Java runtime), in web platforms (Hails [40] and W5 [48]), using software fault isolation (duPro [65]), in smartphones

(Weir [64] and Maxoid [96] for Android phones), in the OS kernel (Asbestos [31], HiStar [101], Flume [49], DStar [100], Silverline [61], TightLip [99], LOMAC [36], and IX [57]), across multiple layers (Laminar [74, 68] within the programming language and the OS), or in a hypervisor (Neon [102] and Xen with Demand Emulation [41]). The compliance systems presented in this thesis differ from these systems in a number of ways.

Generally, information flow control that is either language-based or enforced at language runtimes can offer end-to-end guarantees only when the entire pipeline of applications within the system is written in the respective language. Therefore, such enforcement mechanisms are not suitable for complex data retrieval systems whose software stacks involve many programming languages, libraries, and runtimes. Unlike such systems, **T**HOTH and **S**HAI enforce policies on applications written in any programming language, since they enforce data use policies at the OS level.

OS-level information flow control has been studied extensively in the systems community, with seminal works such as Asbestos, HiStar, and Flume. Asbestos and HiStar are specialized operating systems, whereas **T**HOTH and **S**HAI run on standard operating systems (Linux and FreeBSD, respectively). Non-standard operating systems solutions pose a significant practicality hurdle for adoption in data retrieval systems; porting complex processing pipelines and applications to new operating systems is challenging and operationally costly.

Flume extends standard operating systems with IFC. Architecturally, Flume is close to **T**HOTH but different from **S**HAI. Flume and **T**HOTH both isolate processes using a Linux security module and a userspace reference monitor process. **S**HAI, on the other hand, relies on offline flow analysis and OS capability sandboxes. Moreover, **S**HAI's reference monitor runs within the same process in a (privileged) monitor *lwC*. However, like all other OS-level solutions for IFC (Asbestos, HiStar, Silverline, DStar, TightLip, LOMAC, and IX, Laminar), Flume uses abstract labels as taints. In contrast, the compliance systems in this thesis use a declarative policy language to express the data use policies, and the policies themselves constitute the processes' taints.

Using policies as taints, as opposed to using labels as taints, results in two fundamental differences between the compliance systems presented in this thesis and all other OS-level IFC solutions. First, abstract labels require *trusted* application components. These trusted application components are responsible for *(a)* mapping flow policies to abstract labels, and *(b)* performing data declassification. This is in contrast to **T**HOTH and **S**HAI where policies encode the required confidentiality and integrity requirements, including the declassification conditions. Here, a small reference monitor enforces all access and declassification conditions, and application components are trusted only to install correct policies on ingress and egress nodes. Second, the policies in THOTH and SHAI describe the policy configuration completely. With abstract labels, the policy configuration is implicit in the *code* of the trusted application components that perform data declassification and the policy-to-labels mapping (although mapping can be automated to some extent [30]).

Resin [98] enforces programmer-provided policies on PHP and Python web applications. Unlike our declarative policies, Resin's policies are specified as PHP/Python functions. Resin tracks flows at object granularity. THOTH and SHAI enforce flow policies at process granularity, which matches the pipelined structure of data retrieval systems and reduces overhead significantly.

Hails [40] is a Haskell-based web development framework with statically-enforced IFC. THOTH and SHAI offer IFC at the process boundary, and are independent of any language, runtime, or framework used for developing applications. COWL [81] confines JavaScript browser contexts using labels and IFC. The compliance systems developed in this thesis address the complementary problem of controlling data flows on the server side. Both Hails and COWL use DC-labels [79] as policies. DC-labels cannot express content-dependent policies like our censorship, mandatory access logging and typed declassification policies.

5.4 Declarative Policies

Our policy language is based on Datalog and linear temporal logic (LTL). Datalog and LTL are well-studied foundations for policy languages (see [51, 16, 28, 67] and [13, 14, 38, 12], respectively), known for their clarity, conciseness, and high-level of abstraction. The primary innovation in the policy language is its two-layered structure, where the first layer specifies access policies and the second layer specifies declassification policies.

Some operating systems (Nexus [76] and Taos [94]), file systems (PFS [88] and PCFS [39]), and at least one cyber-physical system (Grey [15]) and storage systems (Guardat [87] and Pesos [47]) enforce access policies expressed in Datalog-like languages. THOTH and SHAI can enforce similar policies but, additionally, they can enforce flow policies and declassification policies that these systems cannot enforce. Like Guardat, but unlike the other systems listed above, our policy language supports content-dependent policies. The design of THOTH's reference monitor is inspired by that of Guardat's. Both are separate user-space processes that intercept I/O to evaluate policies. However, THOTH's reference monitor tracks data flows, supports declassification policies, and intercepts memcached I/O and network communication, all of which Guardat's monitor does not do.

5.5 Hybrid Analysis

SHAI combines offline analysis (which can be viewed as a static analysis) and light-weight runtime monitoring. There are a number of systems that follow a similar approach for information flow control [60, 73], enforcing safety properties [37, 34], and gradual (IFC) typing [18, 35, 77]. Fredrikson et al. in [37] use abstraction refinement and model checking to instrument code with sufficient checks to enforce policies, and Rocha et al. in [73] use code analysis to inject policy checks in program code to enforce IFC and declassification policies. Moore and Chong use static analysis to reduce monitoring overhead by selectively marking variables which cannot impose security violations to not be tracked at runtime [60]. Similar to SHAI, these approaches try to perform as much of the enforcement as possible statically, and use runtime checks where static checks are not possible (i.e, when safety relies on information not available statically).

However, all these systems require the source code of the application, are language-dependent, perform the analysis at *fine granularity* (i.e., program variables), and require re-running the analysis when code changes. In contrast, SHAI's offline analysis uses only a description of the system pipeline, not the source or the compiled code of the application. Therefore, SHAI's offline analysis is language-independent. However, since SHAI combines static and dynamic analysis at coarse-granularity, SHAI's analysis is less permissive than the static (program) analysis used by these systems (since SHAI cannot map program inputs to outputs precisely). Nonetheless, SHAI's analysis is suitable for data retrieval systems where application codebases are large and frequently updated, and often written in different programming languages. In SHAI, application updates that do not change tasks' I/O accesses do not require re-running the offline analysis. Even with I/O changes, not re-running the offline analysis will only increase runtime overhead since accesses will have to be checked at runtime. As far as we know, SHAI is the first system to combine static and dynamic analysis at coarse-granularity, and the first to track meta-data changes that affect the prior decisions of the static analysis and to adjust them accordingly as discussed in Subsection 4.2.3.

RIF [45] is a policy model similar in concept to the policy model of the compliance systems presented in this thesis. RIF policies are automata where states represent restrictions and transitions define allowed changes to restrictions. RIF has been implemented in an extension of the Java programming language called JRIF [46]. Like the aforementioned work, JRIF enforces policies at fine-granularity by hybrid analysis consisting of mostly static inference and some runtime checks. All the differences from SHAI mentioned above apply to JRIF as well. Additionally, JRIF's declassification conditions are linked to specific program points, not predicates on the system/conduit state as in SHAI. It is unclear whether a search engine pipeline can be implemented in JRIF and, if so, what the cost of the runtime checks would be.

5.6 Policy Debugging

EON [19] is a programming logic framework to model and to automatically verify dynamic access control policies. This is useful in finding policy configuration errors. Towards the same goal, PolSim [1] uses flow simulation and coverage testing. Other techniques use model checking [104]. These policy debugging frameworks are orthogonal to (and can complement) the compliance systems presented in this thesis. THOTH and SHAI assume correct policies on data sources and sinks, and flows cannot violate these policies regardless of policy configurations on intermediate conduits, whereas IFC policy debugging systems reason about which flows are permitted (or denied) under a given policy configuration.

Nonetheless, SHAI's offline analysis and PolSim [1] share common techniques. Both simulate data flows, and evaluate associated declarative data use policies. In fact, PolSim runs over policies written in the policy language presented in this thesis. However, both differ in a few design decisions which fit their respective goals. PolSim requires hints to evaluate policies that refer to runtime information, processes' taints represent protection requirements of consumed data, and PolSim can provide suggestions to relax policies responsible for denied flows. On the other hand, SHAI's offline analysis defers policies that cannot be resolved statically to the runtime, taints restrict all possible flows a process might attempt, and denied flows due to policy violations are assumed to be correct policy intent.

6 Concluding Remarks

Data retrieval systems collect, index, and serve heterogeneous data items. Each item may be subject to a potentially distinct data use policy. Today, policy violations can arise due to missing policy checks, bugs, and misconfigurations in any of the application components, which are often fast-evolving, complex, and with huge codebases. Ensuring policy compliance for data retrieval systems is a challenging problem. This thesis addresses this problem and presents efficient mechanisms to enforce data use policies.

Towards this end, we first designed a declarative policy language to express data owners' privacy preferences, the provider's own data-use policy, and legal requirements. Our policies are directly attached to data conduits, and specify access control (read/write) and flow control (declassification) policies.

Taking into consideration practical aspects (low runtime overhead, compatibility with existing systems, and enforcement separate from application code), we presented **T**HOTH, a kernel-level policy compliance system. **T**HOTH enforces policies by tracking and controlling data flows across tasks through kernel I/O interception. **T**HOTH shows that OS-level flow control is possible without relying on trusted application components, and it can enforce rich declassification policies. Using an optimized prototype, we showed that **T**HOTH can be deployed with low overhead for systems that need to sustain a few hundred requests/second/machine. This demonstrates the viability of coarse-grained taint tracking as a basis for policy enforcement in data retrieval systems.

The design of **T**HOTH, nonetheless, has a fundamental limitation. Since all of **T**HOTH's policy enforcement work happens on the critical I/O path, its runtime overhead increases with the rate of application I/O. Therefore, **T**HOTH incurs significant runtime overhead on high throughput data retrieval systems. This, however, offered the key insight in the design of the second compliance system presented in this thesis.

SHAI is a policy compliance system that enforces data use policies with near-zero runtime overhead in the critical request path of serving requests. The key insight behind **S**HAI is to push as much work as possible to the background in order to achieve a streamlined I/O path. Towards this end, SHAI relies on a combination of an offline flow analysis and a light-weight runtime monitoring using an OS capability sandbox. At a high level, the offline analysis determines the compliant data flows based on runtime profiles and current policies. These compliant flows are then translated into sets of capabilities for each of a service's tasks. At runtime, SHAI allows only compliant flows by running tasks in sandboxes and granting each task the respective capability set. Our prototype shows that SHAI incurs near-zero overhead in the critical path, even with high rates of I/O.

6.1 The Evolution from THOTH to SHAI

SHAI is a re-design of THOTH with the goal of mitigating most of THOTH's runtime overhead. As we abstract away from the implementation details of both systems, the main difference between THOTH and SHAI is the addition of an offline phase that performs most of the compliance checking ahead-of-time. At a high level, depending on the accuracy and the completeness of the offline analysis input, the policy enforcement overhead would fall between being proportional to the rate of I/O (i.e., THOTH-like) to being near-zero (i.e., SHAI's goal). If fact, we envision that systems using SHAI would evolve to avoid most of the policy enforcement runtime overhead, as operational data can be used to improve the accuracy and the completeness of the offline analysis input over time.

First, it is safe to argue that SHAI modulo the offline analysis would reduce to THOTH, since the other key difference between the two systems is the different implementation for tasks. Instead of processes, SHAI uses *lwCs* to achieve faster state reset. Using *lwCs* instead of processes, however, is an implementation detail that can easily amend THOTH's implementation. (In fact, our evaluation presented in Section 4.5 uses DYNAMIC which does exactly that.)

Now, with the main difference between SHAI and THOTH reduced to the offline analysis, the completeness and accuracy of the offline analysis input determine whether the enforcement overhead is on par with THOTH-like enforcement or rather near-zero. These two performance points are straightforward to reason about. On one hand, when the offline analysis input is completely wrong, all I/O is intercepted and is subject to runtime policy evaluation. On the other hand, when the offline analysis input perfectly predicts the accesses that happen at runtime, policy enforcement comes at no runtime cost as no I/O is intercepted. In practice, we believe that the policy enforcement overhead using **S**HAI would gradually decrease as the accuracy and completeness of the offline analysis input improve.

Initially, an administrator can manually reason about the expected tasks' taints and the legitimate flows in the system and use such manual analysis to seed the offline analysis input. Such manual analysis can potentially be inaccurate and incomplete. With an offline analysis input that is mostly inaccurate, **S**HAI's policy enforcement would be close to that of **T**HOTH as the majority of I/O will be intercepted and will be subject to runtime policy evaluation, resulting in runtime overhead proportional to the rate of I/O (i.e., **T**HOTH-like). In practice, the starting overhead point for policy enforcement with **S**HAI does not necessarily need to be **T**HOTH-like. In fact, manual analysis can closely approximate the runtime behaviour for simple pipelines (such as Sys-E described in 4.2.1). As the pipeline grows in complexity, manual analysis is likely to introduce mispredictions, resulting in runtime overhead.

Operational data, over time, can improve the quality of the offline analysis input (accuracy and completeness). In principle, there can be automated tools to help system administrators refine the offline analysis input. These tools can use monitoring the system in production, testing and simulation, and analyzing traces and logs in order to improve the predictions of the system's (future) runtime behaviour, moving the performance point of enforcement overhead to near-zero.

7 Future Directions

7.1 Further Engineering in SHAI

In this section, we describe a few optimizations that can reduce the runtime overhead of SHAI even further.

Pool of lwCs for user sessions. Our SHAI prototype relies on light-weight contexts (*lwCs*) to isolate user sessions. Each incoming user connection requires creating a new front-end task *lwC*. This is necessary for the security guarantees of SHAI since reusing *lwCs* across sessions would allow a buggy application to leak the private data of one user to another. However, creating *lwCs* incurs runtime overhead, which becomes significant when the rate of incoming connections is high.

Data retrieval systems whose workloads are characterized by idle periods followed by short bursts of incoming connections may reduce the runtime overhead at peak by maintaining a pool of pre-created *lwCs*. During idle (or under-utilized) periods, the system adds pristine *lwCs* to the pool, which can be used later. Adding support for such features is straightforward and can effectively reduce SHAI's runtime overhead at workload peaks.

Piggybacking on the application's own capability checks. SHAI relies on OS capability sandboxes for its runtime enforcement. Capability checks are generally efficient since they are streamlined in the underlying kernel. However, under extremely high throughput, capability checks can incur non-negligible runtime overhead. For instance, we showed in Subsection 4.5.2 that SHAI's runtime overhead is 1.2% at around 30,000 search request/second/machine (under long sessions). This overhead is due to checking, at a very high rate, that the front-ends in Sys-E have existing capabilities over the transferred file descriptors (preventing a buggy search engine from sending Alice's private file to Bob). We discuss next how this overhead can be reduced when applications anyway perform the necessary access checks.

In principle, in a system like Sys-E, the search engine should check if the user who would receive the file descriptors has access to the documents referenced by them. This check is necessary to avoid data leaks due to a corrupt index. (Filtering search results typically relies on attributes encoded in the index. A corrupt index may associate Alice's private data with Bob.) While our Sys-E implementation does not perform such checks (since *(a)* we wanted to evaluate against a very competitive baseline with as high throughput as possible and *(b)* our threat model allows for missing and incorrect application checks), this check should exist in production systems that follow good engineering practices. With **S**HAI in place, one way to perform such a check is to query the underlying capability system (Capsicum in our prototype) if the capability set of the connected front-end includes the file descriptors to be transferred. If the file descriptors are indeed included, the capability system can mark them at that point as safe to be transferred to the front-end. Then, the capability system does not need to perform the inclusion check when transferring the file descriptors and immediately allow the transfer if they are already marked as safe.

Admittedly, piggybacking on the application's own checks does not improve the throughput of the system: regardless whether **S**HAI is in place or not, the performance of the application will be slower when the application performs access checks. Nonetheless, it is important to note that **S**HAI's relative overhead is minimal when the baseline applications perform the expected access checks, since these checks can offset the cost of **S**HAI's policy enforcement.

7.2 Automatic Bootstrapping of Policy Configuration

The policy compliance systems presented in this thesis assume that ingress and egress policies are correct and are attached to all relevant data sources and sinks. While assigning policies to data in a new data retrieval system as it evolves is not particularly challenging, it might be a colossal undertaking for existing large-scale data retrieval providers: Existing large-scale providers may already have millions of users, billions of data items (replicated and shared across different services) with a complex dependency graph of derived information, and many policies and checks scattered across many large application codebases.

We do not address that problem in this thesis. We acknowledge that, in order for the policy compliance systems developed in this thesis to be adopted by existing large-scale providers, the compliance systems must be accompanied with tools to automate policy configuration. Following Grok's insights, these tools can rely on logs, heuristics, program analysis, and selective manual verification to bootstrap policy configuration in an existing system [75].

7.3 Database-Backed Data Retrieval Systems

We have focused on enforcing data use policies in data retrieval systems that operate on unstructured data. Next, we discuss how the compliance systems presented in this thesis can, in principle, be applied to database-backed systems.

One approach is to intercept all application queries in the database adapter, look up applicable policies, and rewrite queries to ensure compliance. This is the approach adopted by Qapla [58]. Qapla[1] can enforce rich and fine-grained (column-, row-, and cell-level) access control policies specified in SQL. Our experience with Qapla shows that the overhead of processing the (policy-compliant) rewritten queries depends on the complexity of the policies in place.

While Qapla's enforcement is sufficient to enforce access control at the system's edge (i.e., users' reads and writes), data retrieval systems consist of processing pipelines where end-to-end guarantees require flow control. Therefore, *(a)* extending Qapla's policies to express flow control and *(b)* studying how to efficiently enforce such column-, row- and cell-level flow policies are two directions that can broaden the applicability of the compliance systems developed in this thesis. Both directions are challenging. Nonetheless, an approach similar to that of SHAI might be useful for efficient (low-overhead) enforcement: A task's queries are analyzed offline to determine the overall taint applicable to the query results (such taint would be checked against the task's taint for restrictiveness), and the database adapter would allow only queries that were already certified by the offline analysis.

7.4 Beyond Data Retrieval Systems

General data processing systems. In principle, the policy compliance systems presented in this thesis could be applied to more general data processing systems. In contrast to data retrieval systems (whose outputs are a mash-up of the input data and whose computations are used only to select input data for inclusion in the output), general processing systems perform statistical operations (average, summation, etc.) and other rich transformations on input data. The compliance systems presented in this thesis can be extended to enforce policies in such general processing systems by including a richer set of declassification operators (e.g., statistical operators) that can be enforced directly by the reference monitor.

[1]I was involved in the development of Qapla, which was led by my colleague Aastha Mehta.

Processing systems with strict isolation between teams. In addition to the statistical transformation mentioned above, some processing systems are often interested in strict isolation between teams with conflicting interests. Such strict isolation, called the Chinese Wall [17], is often necessary in information systems for finance, audit, insurance, and law firms. SHAI can enforce Chinese Wall policies efficiently: the offline analysis grants distinct sets of capabilities to conflicting teams. Moreover, the policy language, as is, can express various declassification conditions to relax the Chinese Wall restriction as needed (e.g., after a certain date or when a report is finalized).

Bibliography

[1] Mohamed Alzayat. "PolSim: Automatic Policy Validation via Meta-Data Flow Simulation". MA thesis. Saarbruecken: Saarland University, 2016.

[2] Gregory R. Andrews and Richard P. Reitman. "An Axiomatic Approach to Information Flow in Programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1 (Jan. 1980).

[3] Apache Software Foundation. *Apache Lucene*. `http://lucene.apache.org`. Last accessed June 2019.

[4] Apache Software Foundation. *CharFilter offsets correction is wonky*. `https://issues.apache.org/jira/browse/LUCENE-6595`. Apache Lucene bug report. Last accessed June 2019.

[5] Apache Software Foundation. *Field names can be wrong for stored fields / term vectors after merging*. `https://issues.apache.org/jira/browse/LUCENE-3575`. Apache Lucene bug report. Last accessed June 2019.

[6] Apache Software Foundation. *MultiSearcher.rewrite() incorrectly rewrites queries*. `https://issues.apache.org/jira/browse/LUCENE-2756`. Apache Lucene bug report. Last accessed June 2019.

[7] Apache Software Foundation. *Negative wildcard searches on MultiSearcher not eliminating correctly*. `https://issues.apache.org/jira/browse/LUCENE-1300`. Apache Lucene bug report. Last accessed June 2019.

[8] Apache Software Foundation. *QueryWrapperFilter discards the IndexReaderContext when delegating to the wrapped query*. `https://issues.apache.org/jira/browse/LUCENE-6503`. Apache Lucene bug report. Last accessed June 2019.

[9] Apache Software Foundation. *Special Characters inside a query resolve in wrong hits Export*. `https://issues.apache.org/jira/browse/LUCENE-49`. Apache Lucene bug report. Last accessed June 2019.

[10] Apache Software Foundation. *SynonymFilter behaves not as expected with ignoreCase=true*. `https://issues.apache.org/jira/browse/LUCENE-6832`. Apache Lucene bug report. Last accessed June 2019.

[11] Apache Software Foundation. *TermsFilter might return wrong results if a field is not indexed or not present in the index*. `https://issues.apache.org/jira/browse/LUCENE-4511`. Apache Lucene bug report. Last accessed June 2019.

[12] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. "Privacy and Contextual Integrity: Framework and Applications". In: *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*. 2006.

[13] Adam Barth, John C. Mitchell, Anupam Datta, and Sharada Sundaram. "Privacy and Utility in Business Processes". In: *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*. 2007.

[14] David A. Basin, Felix Klaedtke, and Samuel Müller. "Policy Monitoring in First-Order Temporal Logic". In: *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV)*. 2010.

[15] Lujo Bauer, Scott Garriss, and Michael K. Reiter. "Distributed Proving in Access-Control Systems". In: *Proceedings of the 26th IEEE Symposium on Security and Privacy (S&P)*. 2005.

[16] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. "Design and Semantics of a Decentralized Authorization Language". In: *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*. 2007.

[17] David F. C. Brewer and Michael J. Nash. "The Chinese Wall Security Policy". In: *Proceedings of the 10th IEEE Symposium on Security and Privacy (S&P)*. 1989.

[18] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. "HLIO: Mixing Static and Dynamic Typing for Information-flow Control in Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2015.

[19] Avik Chaudhuri, Prasad Naldurg, Sriram K. Rajamani, G. Ramalingam, and Lakshmisubrahmanyam Velaga. "EON: Modeling and Analyzing Dynamic Access Control Systems with Logic Programs". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. 2008.

[20] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. "Abstractions for Usable Information Flow Control in Aeolus". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC)*. 2012.

[21] Adam Chlipala. "Static Checking of Dynamically-varying Security Policies in Database-backed Applications". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2010.

[22] Adam Chlipala. "Ur: Statically-typed Metaprogramming with Type-level Record Computation". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2010.

[23] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. "SubDomain: Parsimonious Server Security". In: *Proceedings of the 14th USENIX Conference on System Administration (LISA)*. 2000.

[24] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. "Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications". In: *Proceedings of the 18th USENIX Security Symposium (USENIX Security)*. 2009.

[25] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*. 2004.

[26] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Communications of the ACM* 19.5 (May 1976).

[27] Dorothy E. Denning and Peter J. Denning. "Certification of Programs for Secure Information Flow". In: *Communications of the ACM* 20.7 (July 1977).

[28] John DeTreville. "Binder, a Logic-Based Security Language". In: *Proceedings of the 23rd IEEE Symposium on Security and Privacy (S&P)*. 2002.

[29] *Dynamic Host Configuration Protocol (DHCP) client*. `https://man.freebsd.org/dhclient`. Last accessed June 2019.

[30] Petros Efstathopoulos and Eddie Kohler. "Manageable Fine-grained Information Flow". In: *Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. 2008.

[31] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. "Labels and Event Processes in the Asbestos Operating System". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. 2005.

[32] Eslam Elnikety, Deepak Garg, and Peter Druschel. "SHAI: Enforcing Data-Specific Policies with Near-Zero Runtime Overhead". In: *CoRR* abs/1801.04565 (2018). URL: `http://arxiv.org/abs/1801.04565`.

[33] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. “Thoth: Comprehensive Policy Compliance in Data Retrieval Systems”. In: *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. 2016.

[34] Úlfar Erlingsson and Fred B. Schneider. “SASI Enforcement of Security Policies: A Retrospective”. In: *Proceedings of the 1999 Workshop on New Security Paradigms (NSPW)*. 2000.

[35] Luminous Fennell and Peter Thiemann. “Gradual Security Typing with References”. In: *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF)*. 2013.

[36] Timothy Fraser. “LOMAC: Low Water-Mark integrity protection for COTS environments”. In: *Proceeding of the 21st IEEE Symposium on Security and Privacy (S&P)*. 2000.

[37] Matthew Fredrikson, Richard Joiner, Somesh Jha, Thomas Reps, Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. “Efficient Runtime Policy Enforcement Using Counterexample-guided Abstraction Refinement”. In: *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*. 2012.

[38] Deepak Garg, Limin Jia, and Anupam Datta. “Policy auditing over incomplete logs: theory, implementation and applications”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*. 2011.

[39] Deepak Garg and Frank Pfenning. “A Proof-Carrying File System”. In: *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*. 2010.

[40] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. “Hails: Protecting Data Privacy in Untrusted Web Applications”. In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012.

[41] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. “Practical Taint-based Protection Using Demand Emulation”. In: *Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems (EuroSys)*. 2006.

[42] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. “Adaptive Parallelism for Web Search”. In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. 2013.

[43] Havard D. Johansen, Eleanor Birrell, Robbert van Renesse, Fred B. Schneider, Magnus Stenhaug, and Dag Johansen. “Enforcing Privacy Policies with Meta-Code”. In: *Proceedings of the 6th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. 2015.

[44] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. "Secure Data Preservers for Web Services". In: *Proceedings of the 2nd USENIX Conference on Web Application Development (WebApps)*. 2011.

[45] Elisavet Kozyri and Fred B. Schneider. *RIF: Reactive Information Flow Labels*. Tech. rep. Cornell University, 2019. URL: `https://ecommons.cornell.edu/handle/1813/65012`.

[46] Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. *JRIF: Reactive Information Flow Control for Java*. Tech. rep. Cornell University, 2016. URL: `https://ecommons.cornell.edu/handle/1813/41194`.

[47] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. "Pesos: Policy Enhanced Secure Object Store". In: *Proceedings of the 13th ACM SIGOPS European Conference on Computer Systems (EuroSys)*. 2018.

[48] Maxwell Krohn, Alex Yip, Micah Brodsky, Robert Morris, and Michael Walfish. "A World Wide Web Without Walls". In: *6th ACM Workshop on Hot Topics in Networking (Hotnets)*. 2007.

[49] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. "Information Flow Control for Standard OS Abstractions". In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 2007.

[50] Leslie Lamport. "The Part-time Parliament". In: *ACM Transactions on Computer Systems* (1998).

[51] Ninghui Li and John C. Mitchell. "Datalog with Constraints: A Foundation for Trust Management Languages". In: *Proceedings of the 5th Symposium on Practical Aspects of Declarative Languages (PADL)*. 2003.

[52] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. "Light-Weight Contexts: An OS Abstraction for Safety and Performance". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016.

[53] Peter Loscocco and Stephen Smalley. "Integrating Flexible Support for Security Policies into the Linux Operating System". In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (ATC)*. 2001.

[54] Petros Maniatis, Devdatta Akhawe, Kevin Fall, Elaine Shi, Stephen McCamant, and Dawn Song. "Do You Know Where Your Data Are? Secure Data Capsules for Deployable Data Protection". In: *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS)*. 2011.

[55] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[56] Paul Marinescu, Chad Parry, Marjori Pomarole, Yuan Tian, Patrick Tague, and Ioannis Papagiannis. "IVD: Automatic Learning and Enforcement of Authorization Rules in Online Social Networks". In: *Proceedings of 38th IEEE Symposium on Security and Privacy (S&P)*. 2017.

[57] M. D. Mcilroy and J. A. Reeds. "Multilevel Security in the UNIX Tradition". In: *Software—Practice and Experience* 22 (1992).

[58] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. "Qapla: Policy compliance for database-backed systems". In: *26th USENIX Security Symposium (USENIX Security)*. 2017.

[59] *Memcached*. `http://memcached.org/`. Last accessed June 2019.

[60] Scott Moore and Stephen Chong. "Static Analysis for Efficient Hybrid Information-Flow Control". In: *2011 IEEE 24th Computer Security Foundations Symposium (CSF)*. 2011.

[61] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. "SilverLine: Preventing Data Leaks from Compromised Web Applications". In: *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*. 2013.

[62] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1999.

[63] Andrew C. Myers and Barbara Liskov. "Protecting Privacy Using the Decentralized Label Model". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.4 (Oct. 2000).

[64] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. "Practical DIFC Enforcement on Android". In: *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. 2016.

[65] Ben Niu and Gang Tan. "Efficient User-space Information Flow Control". In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*. 2013.

[66] *OpenSSH*. `https://www.freebsd.org/doc/handbook/openssh.html`. Last accessed June 2019.

[67] Andrew Pimlott and Oleg Kiselyov. "Soutei, a Logic-Based Trust-Management System". In: *Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS)*. 2006.

[68] Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. Mckinley, and Emmett Witchel. "Practical Fine-Grained Information Flow Control Using Laminar". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.1 (Nov. 2014).

[69] François Pottier and Vincent Simonet. "Information Flow Inference for ML". In: *The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2003.

[70] *Privacy Rights Clearinghouse*. `http://privacyrights.org`. Last accessed June 2019.

[71] Reuters. *Adobe data breach more extensive than previoulsy disclosed*. `http://www.reuters.com/article/2013/10/29/us-adobe-cyberattack-idUSBRE99S1DJ20131029`. Last accessed June 2019.

[72] Reuters. *Target breach worse than thought, states launch joint probe*. `http://www.reuters.com/article/2014/01/10/us-target-breach-idUSBREA090L120140110`. Last accessed June 2019.

[73] Bruno P. S. Rocha, Mauro Conti, Sandro Etalle, and Bruno Crispo. "Hybrid Static-Runtime Information Flow and Declassification Enforcement". In: *IEEE Trans. Information Forensics and Security* 8.8 (2013).

[74] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. "Laminar: Practical Fine-grained Decentralized Information Flow Control". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2009.

[75] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. "Bootstrapping Privacy Compliance in Big Data Systems". In: *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*. 2014.

[76] Alan Shieh, Dan Williams, Emin Gün Sirer, and Fred B Schneider. "Nexus: a new operating system for trustworthy computing". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*. 2005.

[77] Jeremy G. Siek and Walid Taha. "Gradual Typing for Functional Languages". In: *In Scheme and Functional Programming Workshop*. 2006.

[78] Daniel J. Solove and Woodrow Hartzog. "The FTC and the New Common Law of Privacy". In: *Columbia Law Review* 114 (2014).

[79] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. "Disjunction category labels". In: *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications (NordSec)*. 2011.

[80] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. "Flexible Dynamic Information Flow Control in Haskell". In: *Proceedings of the 4th ACM Symposium on Haskell*. 2011.

[81] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. "Protecting Users by Confining JavaScript with COWL". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014.

[82] *The EU General Data Protection Regulation*. `https://www.eugdpr.org/the-regulation`. Last accessed June 2019.

[83] The Guardian. *Facebook says nearly 50m users compromised in huge security breach*. `https://www.theguardian.com/technology/2018/sep/28/facebook-50-million-user-accounts-security-berach`. Last accessed June 2019.

[84] The New York Times. *Facebook Security Breach Exposes Accounts of 50 Million Users*. `https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach.html`. Last accessed June 2019.

[85] *The Smack Project*. `http://schaufler-ca.com`. Last accessed June 2019.

[86] *TOMOYO Linux*. `http://tomoyo.osdn.jp/index.html.en`. Last accessed June 2019.

[87] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. "Guardat: Enforcing data policies at the storage layer". In: *Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. 2015.

[88] Kevin Walsh and Fred B. Schneider. *Costs of Security in the PFS File System*. Tech. rep. Computing and Information Science, Cornell University, 2012.

[89] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. "A Taste of Capsicum: Practical Capabilities for UNIX". In: *Commununications of the ACM* 55.3 (Mar. 2012).

[90] Wikimedia Foundation. *Image Dump*. `http://archive.org/details/wikimedia-image-dump-2005-11`. Last accessed June 2019.

[91] Wikimedia Foundation. *Static HTML dump*. `http://dumps.wikimedia.org/`. Last accessed June 2019.

[92] Wikipedia. *Data breach: Major incidents*. `http://en.wikipedia.org/wiki/Data_breach#Major_incidents`. Last accessed June 2019.

[93] Wired. *Reporters sued as hackers for finding a security hole with Google*. https://www.wired.co.uk/article/reporter-google-breach-hacker. Last accessed June 2019.

[94] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. "Authentication in the Taos Operating System". In: *ACM Transactions on Computer Systems (TOCS)* 12.1 (1994).

[95] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. "Linux Security Modules: General Security Support for the Linux Kernel". In: *Proceedings of the 11th USENIX Security Symposium (USENIX Security)*. 2002.

[96] Yuanzhong Xu and Emmett Witchel. "Maxoid: Transparently Confining Mobile Applications with Custom Views of State". In: *Proceedings of the 10th ACM SIGOPS European Conference on Computer Systems (EuroSys)*. 2015.

[97] Dawei Yin, Yuening Hu, Jiliang Tang, Tim Daly, Mianwei Zhou, Hua Ouyang, Jianhui Chen, Changsung Kang, Hongbo Deng, Chikashi Nobata, Jean-Marc Langlois, and Yi Chang. "Ranking Relevance in Yahoo Search". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2016.

[98] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. "Improving application security with data flow assertions". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. 2009.

[99] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. "TightLip: Keeping Applications from Spilling the Beans". In: *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*. 2007.

[100] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. "Securing Distributed Systems with Information Flow Control". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2008.

[101] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making Information Flow Explicit in HiStar". In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006.

[102] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. "Neon: System Support for Derived Data Management". In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 2010.

[103] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. "Using CQUAL for Static Analysis of Authorization Hook Placement". In: *Proceedings of the 11th USENIX Security Symposium (USENIX Security)*. 2002.

[104] Mingyi Zhao and Peng Liu. "Modeling and Checking the Security of DIFC System Configurations". In: *Automated Security Management*. Ed. by Ehab Al-Shaer, Xinming Ou, and Geoffrey Xie.

List of Figures

List of Tables

List of Algorithms

A Policies for Data Flows in a Search Engine

In this appendix, we provide details of the policies used in our policy-compliant search engine. All policies are represented in the **read**, **update** and **declassify** rules on source conduits (documents that the search engine indexes, the user profile, etc.). We describe these rules incrementally: We start from a set of base rules, which we refine to include more policies.

Base rules Our base rules allow anyone to read, update or destroy the source conduit they are attached to.

read :- $TRUE$
update :- $TRUE$
destroy :- $TRUE$
declassify :- isAsRestrictive(**read**, `this`.**read**)
  until $FALSE$

The **read**, **update** and **destroy** rules have condition $TRUE$, which always holds, so these rules do not restrict access at all. The **declassify** rule insists that the **read** rule on any conduit containing data derived from the source conduit be at least as restrictive as the **read** rule above, which will always be the case (because the **read** rule above is the most permissive read rule possible). This base policy is pointless in itself, but it serves as the starting point for the remaining policies.

A.1 Client Policies

First, we describe policies to represent client privacy preferences.

A.1.1 Private policy

A user Alice may wish that her private files (e.g., her e-mails) be accessible only to her. This can be enforced by requiring that accesses to Alice's private files happen in the context of a session authenticated with Alice's key. Technically, this is accomplished by replacing the conditions in the base **read**, **update** and **destroy** rules as shown below and attaching the resulting rules to Alice's private files. The predicate sKeyIs(k) means that the current session is authenticated using the public key k.

read :- sKeyIs($k_{\texttt{Alice}}$)
update :- sKeyIs($k_{\texttt{Alice}}$)
destroy :- sKeyIs($k_{\texttt{Alice}}$)

The **declassify** rule remains unchanged. It ensures that any conduit containing data derived from Alice's private files is subject to a **read** rule that is at least as restrictive as the revised **read** rule above. Hence, no such conduit can be read by anyone other than Alice.

A.1.2 Friends-only policy

Alice might want that her blog and online social network profile be readable by her friends. To do this, she could add a disjunctive ("or"-separated) clause in the read rule requiring that read accesses happen in the context of a session authenticated with a key k_X of one of Alice's friends. Alice's friends are assumed to be listed in the file Alice.acl, which contains an entry of the form isFriend(k_X, X_{ACL}) for each public key k_X that belongs to a friend of Alice. The isFriend entry also states the file X_{ACL} which lists the friends of the key k_X's owner. Note that the isFriend entry format presented in Section 3.2 was slightly simplified for readability.

read :- sKeyIs($k_{\texttt{Alice}}$) $\vee$
[sKeyIs(k_X) $\wedge$ ("Alice.acl", *off*) says isFriend(k_X, X_{ACL})]

The predicate (('Alice.acl", *off*) says isFriend(k_X, X_{ACL})) checks that k_X exists in the list of Alice's friends (file "Alice.acl") at some offset *off*.

A.1.3 Friends-of-friends policy

To additionally allow read access to friends of friends, the policy would require read accesses to happen in the context of an authenticated session whose key is present in the friend list of any of Alice's friends.

$$\begin{array}{l}\textbf{read} \text{ :- } \mathsf{sKeyIs}(k_{\texttt{Alice}}) \lor \\ [\mathsf{sKeyIs}(k_X) \land (\text{"Alice.acl"}, \mathit{off}) \text{ says } \mathsf{isFriend}(k_X, X_{ACL})] \\ \quad \lor \\ [\mathsf{sKeyIs}(k_Y) \land (\text{"Alice.acl"}, \mathit{off_1}) \text{ says } \mathsf{isFriend}(k_X, X_{ACL}) \\ \land \ (X_{ACL}, \mathit{off_2}) \text{ says } \mathsf{isFriend}(k_Y, Y_{ACL})]\end{array}$$

The predicate $((\text{'Alice.acl"}, \mathit{off_1}) \text{ says } \mathsf{isFriend}(k_X, X_{ACL}))$ checks that k_X exists in the list of Alice's friends (file "Alice.acl") at some offset *off*$_1$. It also binds the variable X_{ACL} to the friend list of the key k_X's owner. Next, the predicate $((X_{ACL}, \mathit{off_2}) \text{ says } \mathsf{isFriend}(k_Y, Y_{ACL}))$ checks that the public key that authenticated the session k_Y exists in the list of friends for the k_X's owner at some offset *off*$_2$.

A.2 Provider Policies

Next, we describe two policies that a provider may wish to impose, possibly to comply with legal requirements.

A.2.1 Mandatory access logging (MAL)

The MAL policy allows an authorized employee of the provider read access to a source conduit F if the access is logged. The log entry must have been previously written to the file k.log, where k is the public key of the employee. The log entry must mention the employee's key, the ID of the accessed conduit and the time at which the conduit is accessed with a tolerance of 60 seconds. To enforce these requirements, a new disjunctive condition is added to the last **read** rule above. The ... in the rule below abbreviate the conditions of the last **read** rule above.

$$\begin{array}{l}\textbf{read} \text{ :- } \ldots \lor \\ \quad \mathsf{sKeyIs}(k) \land \mathsf{cIdIs}(F) \land \\ \quad (\text{"auth\_employees"}, \mathit{off}) \text{ says } \mathsf{isEmployee}(k) \land \\ \quad (\text{LOG}_k = \mathsf{concat}(k, \text{".log"})) \land \\ \quad (\text{LOG}_k, \mathit{off_1}) \text{ says } \mathsf{readLog}(k, F, T) \land \mathsf{timeIs}(curT) \land \\ \quad \mathsf{gt}(curT, T) \land \mathsf{sub}(diff, curT, T) \land \mathsf{lt}(diff, 60)\end{array}$$

The predicate $\mathsf{sKeyIs}(k)$ binds the public key that authenticated the session (i.e., the public key of the employee) to the variable k, and $\mathsf{cIdIs}(F)$ binds the name of source conduit to F. Next, the predicate $((\text{"auth\_employees"}, \mathit{off}) \text{ says } \mathsf{isEmployee}(k))$ checks that k exists in the list of authorized employees (file "auth_employees") at some offset *off*, to verify that the source conduit's reader is really an employee. Next, LOG$_k$ is bound to the name of the employee's log file, k.log. The predicate $((\text{LOG}_k, \textit{off}_1)$ says $\mathsf{readLog}(k, F, T))$ checks that the log file contains an appropriate entry with some time stamp T and the remaining predicates check that the current time, $curT$, satisfies $T \leq curT \leq T + 60\text{s}$.

Every log file has a **read** rule that allows only authorized auditors to read the file (the public keys of all authorized auditors are assumed to be listed in the file "auditors"). It also has an **update** rule that allows appends only, thus ensuring that a log entry cannot be removed or overwritten.

read :- sKeyIs(k) $\wedge$ ("auditors", *off*) says isAuditor(k)
update :- sKeyIs(k) $\wedge$
("auth_employees", *off*) says isEmployee(k) $\wedge$
cCurrLenIs($cLen$) $\wedge$ cNewLenIs($nLen$) $\wedge$
gt($nLen, cLen$) $\wedge$ (this$, 0, cLen$) hasHash (h) $\wedge$
(this$, 0, cLen$) willHaveHash (h)

In the append-only policy (rule **update** above), the predicate cCurrLenIs($cLen$) binds the current length of the log file to $cLen$ and the predicate cNewLenIs($nLen$) binds the new length of the log file to $nLen$. Next, the predicate gt($nLen, cLen$) ensures that the update only increases the log file's length. *(c, off, len)* hasHash (or willHaveHash) is a special mode of using says (or willsay) which allows the policy interpreter to refer to the hash of the conduit *c*'s content (or updated content in a write transaction) from offset *off* with length *len*. In the **update** rule, hasHash and willHaveHash are used to verify that the existing file content is not modified during an update by checking that the hashes of the file from offset 0 to *cLen*, originally and after the prospective update, are equal.

A more efficient implementation of the append-only policy could rely on a specialized predicate unmodified(*off*, *len*), which checks that the conduit contents from offset *off* with length *len* were not modified. The **update** rule could then be simplified to:

update :- sKeyIs(k) $\wedge$
("auth_employees", *off*) says isEmployee(k) $\wedge$
cCurrLenIs($cLen$) $\wedge$ cNewLenIs($nLen$) $\wedge$
gt($nLen, cLen$) $\wedge$ unmodified($0, cLen$)

A.2.2 Region-based censorship

Legal requirements may force the provider to blacklist certain source files in certain regions. Accordingly, the goal of the censorship policy is to ensure that content from a document F can only reach users in regions whose blacklists do not contain F. The policy relies on a mapping from IP addresses to regions and a per-region blacklist file. The blacklist file is maintained in a sorted order to efficiently lookup whether it contains a given document or not.

The censorship policy is expressed by modifying the **declassify** rule of every source conduit cndID as follows:

$$\begin{array}{l}\textbf{declassify} :\text{-}\ \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read})\ \mathsf{until} \\ \ (\text{CENSOR}(\mathsf{cndID}) \wedge \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read}))\end{array}$$

The rule says that the **read** rule on any conduit to which cndID flows must be as restrictive as cndID's **read** rule *until* a conduit at which the condition CENSOR(cndID) holds is reached. CENSOR(cndID) is a macro defined below. The predicate $\mathsf{sIpIs}(IP)$ checks that the IP address of the connecting (remote) party is IP and the predicate $\mathsf{IpPrefix}(IP, R)$ means that IP belongs to region R. The blacklist file for region R is R.BlackList. In words, CENSOR(cndID) means that the remote party's IP belongs to a region R and cndID lies strictly between two two consecutive entries in R's blacklist file (and, hence, cndID does not exist in R's blacklist file).

$$\begin{array}{l}\mathsf{sIpIs}(IP) \wedge \mathsf{IpPrefix}(IP, R) \wedge \\ (F_{BL} = \mathsf{concat}(R, \text{".BlackList"})) \wedge \\ (F_{BL}, \mathit{off}_1)\ \mathsf{says}\ \mathsf{isCensored}(cnd_1) \wedge \\ \mathsf{add}(\mathit{off}_2, \mathit{off}_1, \mathsf{CENSOR\_ENTRY\_LEN}) \wedge \\ (F_{BL}, \mathit{off}_2)\ \mathsf{says}\ \mathsf{isCensored}(cnd_2) \wedge \\ \mathsf{lt}(cnd_1, \mathsf{cndID}) \wedge \mathsf{lt}(\mathsf{cndID}, cnd_2)\end{array}$$

A.3 Search Engine Flows

A.3.1 Indexing flow

The indexer reads documents with possibly contradictory policies and, in the absence of a dedicated provision for declassification, the index (and any documents derived from it) cannot be served to any client. To prevent this problem, searchable documents allow typed declassification. The **declassify** rule for each searchable document is modified with a new clause that allows complete declassification into an (internal) conduit whose **update** rule allows the conduit to contain only a list of object ids. The modified **declassify** rule of each source document has the form:

$$\begin{aligned}&\textbf{declassify} \text{ :- } \ldots \text{ until } (\ldots \ \lor (\mathsf{clsIntrinsic} \land \\ &\quad \mathsf{isAsRestrictive}(\textbf{update}, \mathsf{ONLY\_CND\_IDS})))\end{aligned}$$

The macro ONLY_CND_IDS stipulates that only a list of valid conduit ids can be written and it expands to:

$$\begin{aligned}&\mathsf{cCurrLenIs}(\texttt{cLen}) \land \mathsf{cNewLenIs}(\texttt{nLen}) \land \\ &\mathsf{each\ in}(\mathsf{this}, \texttt{cLen}, \texttt{nLen})\ \mathsf{says}(\texttt{cndId}) \\ &\quad \{\mathsf{cIdExists}(\texttt{cndId})\}\end{aligned}$$

In the macro above, the predicate cNewLenIs$(nLen)$ binds the new length of the output file to $nLen$. The predicate willsay checks that the content update from offset 0 and length $nLen$ is a list of conduit IDs, and the predicate cIdExists$(cndId)$ checks that $cndId$ corresponds to an existing conduit.

So far we have assumed that the conduit ids are not themselves confidential. If the presence or absence of a particular conduit id in the search results may leak sensitive information, then the source declassification policy can be augmented to require that the list of conduit ids is accessible only to a principal who satisfies the confidentiality policies of all listed conduits. Then, the macro ONLY_CND_IDS can be re-written to:

$$\begin{aligned}&\mathsf{cCurrLenIs}(\texttt{cLen}) \land \mathsf{cNewLenIs}(\texttt{nLen}) \land \\ &\mathsf{each\ in}(\mathsf{this}, \texttt{cLen}, \texttt{nLen})\ \mathsf{willsay}(\texttt{cndId}) \\ &\{\mathsf{cIdExists}(\texttt{cndId}) \land \mathsf{hasPol}(\texttt{cndId}, \texttt{P}) \land \\ &\ \mathsf{isAsRestrictive}(\textbf{read}, \texttt{P}.\textbf{read}) \land \\ &\ \mathsf{isAsRestrictive}(\textbf{declassify}, \texttt{P}.\textbf{declassify})\}\end{aligned}$$

Additionally in the macro above, the predicate hasPol(`cndId`, `P`) binds `P` to the policy of the conduit `cndId`, and the predicate isAsRestrictive(**read**, `P`.**read**) requires that the confidentiality of the list of conduit ids is as restrictive as the confidentiality requirements of the source conduit ids themselves.

A.3.2 Profile aggregation flow

Since raw user activity logs are typically private, a declassification is required that enables a profile generator to produce a user preferences vector (a vector of fixed length) from the activity logs. However, this preferences vector must further be restricted so that it can be used to produce only a list of conduit ids (the search results). Further, the user might also want to ensure that only activity logs generated in the past 48 hours be used for personalization. This can be achieved by allowing the declassification into the fixed-size vector to happen only within 172800 seconds of the log's creation. Suppose an activity log is created at time t and that the preferences vector has length n. Then, the relevant policy rules on the activity log are the following (note that t and n are constants, not variables).

$$\begin{array}{l}
\textbf{read} \text{ :- } \mathsf{sKeyIs}(k_{\texttt{Alice}}) \\
\textbf{declassify} \text{ :- } [\mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read}) \text{ until} \\
\quad \mathsf{isAsRestrictive}(update, \text{ONLY\_FLOATS}(n)) \wedge \\
\quad \mathsf{clsIntrinsic} \wedge \mathsf{timeIs}(curT) \wedge \mathsf{gt}(curT, t) \wedge \\
\quad \mathsf{sub}(diff, curT, t) \wedge \mathsf{lt}(diff, 172800)] \wedge \\
\quad [\mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read}) \text{ until } \mathsf{clsIntrinsic} \wedge \\
\quad \mathsf{isAsRestrictive}(\textbf{update}, \text{ONLY\_CND\_IDS})]
\end{array}$$

This policy ensures that the raw user logs can only be transformed into the user preferences vector, which in turn can only be declassified into the search results of the search engine.

The macro ONLY_FLOATS(n) stipulates that only a vector of n floats can be written. It expands to:

$$\begin{array}{l}
\mathsf{cNewLenIs}(\texttt{nLen}) \wedge \\
\mathsf{each\ in}(\mathsf{this}, 0, \texttt{nLen}) \ \mathsf{willsay}(\texttt{value}) \\
\{\mathsf{vType}(\texttt{value}, \text{FLOAT}) \wedge (\texttt{Cnt}++)\} \wedge \\
\mathsf{eq}(\texttt{Cnt}, n)
\end{array}$$

In the macro above, the predicate cNewLenIs($nLen$) binds the new length of the output file to $nLen$. The predicate willsay checks that the content update from offset 0 and length $nLen$ is a list of $value$s, and the predicate vType($value$, FLOAT) checks that each $value$ in the list is of type FLOAT. The predicate eq(cnt, n) checks that the update contains n floats.

B Flow Description Language

In this appendix, we describe **S**HAI's flow description language to specify the inputs of the offline analysis (OA). Recall that the OA takes the following inputs:

1. A list of tasks on which to run the OA. If a task's accesses depend on runtime parameters, a separate instance of the task should be listed for every combination of these parameters.
2. For each task, lists of conduits whose reads and writes by this task have to be checked.
3. The steady-state taint of each task.
4. The policies of all conduits in the system.
5. Any policy-relevant meta-data such as friends lists and region-specific content blacklists.

The OA has access to inputs (4) and (5) from the system's meta-data (i.e., policy store and underlying storage systems). Therefore, the flow description language is concerned with inputs (1)–(3). The language has two rules: a *task rule* to specify the system's tasks and their taints, and a *data flow graph rule* to describe expected data flows between tasks and conduits.

Tasks and their taint. The task rule has the form *t :- (id, taint)*, where *id* is a unique identifier for a task and *taint* is the task's taint. Taint is specified in our policy language syntax.

The encoding of this rule supports a simple form of templates. Here, the task rule is extended to *t :- (id*_**[K]**, *taint*_**[var...]**), where each key **K** provides substitutions for a variable list in *taint*. This encoding allows for a concise specification of tasks' taints that encode a combination of parameters (e.g., users identity and regions).

Flows between tasks and conduits. A data flow graph rule has the form *dfg :- (src, dest)*, where *src* and *dest* represent the flow's source and destination, respectively. Exactly one (either *src* or *dest*) is a conduit id, and the other is a task's id.

The encoding of this rule supports two simple types of pattern matching:

- The conduit id can be a directory pathname or a key range, which indicates that the rule applies to all individual conduits contained in the (sub)directories or in the key range. This encoding is useful when, for instance, a task is expected to access many conduits within a specific directory.
- The task id can be a task's id prefix, which indicates that the rule applies to all tasks whose id match the prefix. This encoding is useful when multiple tasks are expected to perform similar flows (e.g., users' tasks may consume all public content).

B.1 Example Data Flow Specification

In this section, we describe the data flow specification in the data retrieval system Sys-E we described in 4.2.1.

LISTING B.1: Task rules in Sys-E

```
1 #The indexer task has the taint INDEX_TAINT
2 Indexer, INDEX_TAINT
3
4 #The search engine task has the taint INDEX_TAINT
5 SearchEngine, INDEX_TAINT
6
7 #Front-end worker tasks has the taint FE_TAINT
8 FrontEnd_[UserInfo:ID], FE_TAINT[UserInfo:PK, UserInfo:Region]
```

Tasks and taints in Sys-E. Listing B.1 shows the task rules in effect in Sys-E. The indexer and the search engine tasks (lines 2 and 5) have the taint INDEX_TAINT, which is the index files policy:

$$\textbf{declassify} \text{ :- } FALSE \text{ until } (\mathsf{clsIntrinsic} \wedge \mathsf{isAsRestrictive}(\textbf{update}, \mathsf{ONLY\_FD+}))$$

With such policy as taint, a task will be able to consume all searchable content and index files that permit the ONLY_FD+ declassification. The ONLY_FD+ shares the same spirit as the ONLY_CND_ID+ presented earlier in Section A.3. ONLY_CND_ID+ ensures that the output is a list of conduit ids and that the destination conduit's policy is at least as restrictive as all the policies of the produced conduit ids. On the other hand, ONLY_FD+ refers to the underlying capability system: It ensures that the output is a list of file descriptors and the destination task already has access capabilities on all conduits referenced by the transferred file descriptors.

Next, the task rule for the front-end workers (line 8) is a template rule. We will consider how this rule is instantiated shortly, but beforehand, let's first consider the (uninstantiated) FE_TAINT, which has the form:

$$\begin{aligned}&\textbf{declassify} \text{ :- } \mathit{FALSE} \text{ until}\\ &\quad ((\mathsf{clsExtrinsic} \wedge \mathsf{sRegionIs}(\mathit{Region})) \wedge \mathsf{isAsRestrictive}(\textbf{read}, \mathsf{sKeyIs}(\mathit{PK})) \vee\\ &\quad (\mathsf{clsIntrinsic} \wedge \mathsf{isAsRestrictive}(\textbf{update}, \text{ONLY\_FD+})))\end{aligned}$$

This taint allows external access to a session connected from geographic region $Region$ and authenticated with public key PK. (It also allows declassification into a list of file descriptors subject to ONLY_FD+.) Both PK and $Region$ are variables which are bound to literals when instantiating the front-end task rules. The instantiation depends on USERINFO, a simple substitution map: ID $\rightarrow$ (PK,REGION). For instance, the rule can be instantiated to Alice's front-end worker taint (i.e., FrontEnd_Alice) using public key $Alice_{pk}$ and region R when USERINFO maps Alice $\rightarrow$ ($Alice_{pk}$, R). USERINFO should include a distinct mapping entry for every pair of PK and $Region$ that the OA should consider for the front-end tasks.

Data flows in Sys-E. Listing B.2 shows the data flow graph rules in effect in Sys-E. The indexer task consumes all searchable content and updates the index files (lines 3 and 4, respectively). In our setup, all searchable contents are included in the directory `/etc/media/content`, and all index files are in the directory `/etc/media/index`.

Next, the search engine consumes the index files (line 8), consumes search queries sent by front-end workers over query pipes under `/etc/media/querypipes` (line 9), and may open all searchable content to later transfer file descriptors to front-end workers (line 10).

Finally, the front-end workers submit queries to the search engine (line 15) and may open all searchable content (line 16). Note that the later is an over-approximation: The OA checks the policies on the searchable content conduits against the front-end workers' taint and certifies only those accesses that are policy compliant. The task id in the data flow graph rules on lines 15 and 16 is a prefix. This prefix is matched against all tasks' ids, and the rule is instantiated for all matches. (For instance, FrontEnd_* matches FrontEnd_Alice included in the task rules.)

LISTING B.2: Flow rules in Sys-E

```
#The indexer (i) consumes all searchable content and
#(ii) updates the the index files
/etc/media/content, Indexer
Indexer, /etc/media/index

#The search engine (i) consumes the index files, (ii) consumes
#the search queries, and (iii) may open all searchable content
/etc/media/index, SearchEngine
/etc/media/querypipes, SearchEngine
/etc/media/content, SearchEngine

#The front-end worker of a user (i) submits search queries to
#the search engine and (ii) possibly consumes all searchable
#content (over-approximation)
FrontEnd_*, /etc/media/querypipes
/etc/media/content, FrontEnd_*
```

The flows in Listing B.2 do not capture the file descriptors transfer from the search engine to the front-end workers since such transfer happens over a socket created at runtime. However, before the socket is established at runtime, SHAI evaluates the search engine's taint to check if it is safe to permit such socket. When the search engine has the taint INDEX_TAINT described earlier, only a socket that has no read and no write permissions is allowed (i.e., a socket that cannot carry data and can only be used to transfer file descriptors).